Cours pour apprendre à utiliser le framework Laravel 5.5

Les bases

Par Maurice Chavelli

Date de publication : 18 avril 2018

Ce tutoriel vous présente la version 5.5 du framework Laravel. C'est un cours qui va vous apprendre les bases de ce framework.

Commentez

I - Présentation générale 5
I-A - Un framework ? 5
I-A-1 - Approche personnelle 5
I-A-2 - (Re)découvrir PHP 5
I-A-3 - Un framework 5
I-B - Pourquoi Laravel ? 6
I-B-1 - Constitution de Laravel 6
I-B-2 - Le meilleur de PHP 6
I-C - La documentation 7
I-D - MVC ? POO ? 7
I-D-1 - MVC 7
I-D-2 - POO 8
I-E - En résumé 8
II - Un environnement de développement 8
II-A - Laragon 9
II-A-1 - Installation 9
II-A-2 - Hôte virtuel 10
II-B - Composer 11
II-B-1 - Présentation 11
II-C - JSON 11
II-C-1 - Packagist 12
II-C-2 - Packalyst 12
II-D - Les éditeurs de code 12
II-E - En résumé 13
III - Installation et organisation 13
III-A - Créer une application Laravel 14
III-A-1 - Le serveur 14
III-A-2 - Prérequis 14
III-A-3 - Installation avec Composer 14
III-A-4 - Installation avec Laravel Installer 15
III-A-5 - Installation avec Laragon 16
III-A-6 - Autorisations 16
III-B - Des URL propres 16
III-C - Organisation de Laravel 17
III-C-1 - Dossier app 17
III-C-2 - Autres dossiers 17
III-C-3 - Fichiers de la racine 18
III-C-4 - Accessibilité 18
III-D - Environnement et messages d'erreur 19
III-E - Une application d'exemple 21
III-F - En résumé 21
IV - Le routage 21
IV-A - Les requêtes HTTP 21
IV-A-1 - Petit rappels 21
IV-A-2 - Les méthodes 23
IV-A-3 - .htaccess et index.php 23
IV-A-4 - Le cycle de la requête 24
IV-B - Plusieurs routes et paramètres de route 25
IV-C - Erreur d'exécution et contrainte de route 26
IV-D - Route nommée 27
IV-E - Ordre des routes 27
IV-F - En résumé 28
V - Les réponses 28
V-A - Les réponses automatiques 28
V-B - Construire une réponse 30
V-C - Les vues 31
V-C-1 - Vue paramétrée 32
V-C-2 - Route 33

V-C-3 - Vue....33
V-D - Blade....34
V-E - Un template....35
V-F - Les redirections....38
V-G - En résumé....38
VI - Artisan et les contrôleurs....38
VI-A - Artisan....39
VI-B - Les contrôleurs....40
VI-B-1 - Rôle....40
VI-B-2 - Constitution....41
VI-B-3 - Liaison avec les routes....42
VI-B-4 - Route nommée....43
VI-C - Utilisation d'un contrôleur....43
VI-D - En résumé....45
VII - Formulaires et middlewares....45
VII-A - Scénario et routes....45
VII-B - Les middlewares....46
VII-C - Le formulaire....47
VII-D - Laravel Collective ?....48
VII-E - Le contrôleur....48
VII-F - La protection CSRF....50
VII-G - Page d'erreur personnalisée....51
VII-H - En résumé....53
VIII - La validation....53
VIII-A - Scénario et routes....54
VIII-A-1 - Routes....54
VIII-B - Les vues....55
VIII-B-1 - Le template....55
VIII-B-2 - La vue de contact....55
VIII-B-3 - Les messages en français....57
VIII-B-4 - La vue de confirmation....58
VIII-C - La requête de formulaire....59
VIII-D - Le contrôleur....60
VIII-E - D'autre façons d'effectuer la validation....61
VIII-F - En résumé....62
IX - Envoyer un email....63
IX-A - Le scénario....63
IX-B - Configuration....64
IX-C - La classe Mailable....64
IX-D - Transmission des informations à la vue....65
IX-D-1 - Test de l'email....67
IX-E - Envoyer des emails en phase développement....67
IX-E-1 - Le mode Log....67
IX-E-2 - MailTrap....68
IX-F - En résumé....70
X - Configuration, session et gestion de fichiers....70
X-A - La configuration....70
X-B - Les sessions....71
X-C - La gestion des fichiers....72
X-D - La requête de formulaire....73
X-E - Les routes et le contrôleur....74
X-F - Les vues....76
X-G - En résumé....78
XI - Injection de dépendance, conteneur et façades....78
XI-A - Le problème et sa solution....79
XI-A-1 - Le problème....79
XI-A-2 - La solution....80
XI-B - La gestion....81

XI-C - Les façades..... 83
XI-D - En résumé..... 84

I - Présentation générale

Dans ce premier chapitre je vais évoquer PHP, son historique rapide et sa situation actuelle. Je vais aussi expliquer l'intérêt d'utiliser un framework pour ce langage et surtout pourquoi j'ai choisi Laravel. J'évoquerai enfin le patron MVC et la Programmation Orientée Objet.

I-A - Un framework ?

I-A-1 - Approche personnelle

PHP est un langage populaire et accessible. Il est facile à installer et présent chez tous les hébergeurs. C'est un langage riche et plutôt facile à aborder, surtout pour quelqu'un qui a déjà des bases en programmation. On peut réaliser rapidement une application web fonctionnelle grâce à lui. Mais le revers de cette simplicité est que bien souvent le code créé est confus, complexe, sans aucune cohérence. Il faut reconnaître que PHP n'encourage pas à organiser son code et rien n'oblige à le faire.

Lorsqu'on crée des applications PHP on finit par avoir des routines personnelles toutes prêtes pour les fonctionnalités récurrentes, par exemple pour gérer des pages de façon dynamique. Une fois qu'on a créé une fonction ou une classe pour réaliser une tâche il est naturel d'aller la chercher lorsque la même situation se présente. Puisque c'est une bibliothèque personnelle et qu'on est seul maître à bord il faut évidemment la mettre à jour lorsque c'est nécessaire, et c'est parfois fastidieux.

En général on a aussi une hiérarchie de dossiers à laquelle on est habitué et on la reproduit quand on commence le développement d'une nouvelle application. On se rend compte parfois que cette habitude a des effets pervers parce que la hiérarchie qu'on met ainsi en place de façon systématique n'est pas forcément la plus adaptée.

En résumé l'approche personnelle est plutôt du bricolage à la hauteur de ses compétences et de sa disponibilité.

I-A-2 - (Re)découvrir PHP

Lorsque j'ai découvert PHP à la fin du dernier millénaire (ça fait plus impressionnant dit comme ça) il en était à la version 3. C'était essentiellement un langage de script en général mélangé au HTML qui permettait de faire du templating, des accès aux données et du traitement. La version 4 en 2000 a apporté plus de stabilité et une ébauche de l'approche objet. Mais il a fallu attendre la version 5 en 2004 pour disposer d'un langage de programmation à la hauteur du standard existant pour les autres langages.

Cette évolution incite à perdre les mauvaises habitudes si on en avait. Un site comme **http://www.phptherightway.com** offre des pistes pour mettre en place de bonnes pratiques. Donc si vous êtes un bidouilleur de code PHP je vous conseille cette saine lecture qui devrait vous offrir un nouvel éclairage sur ce langage et surtout vous permettre de vous lancer de façon correcte dans le code de Laravel.

I-A-3 - Un framework

D'après Wikipedia un framework informatique est un « ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel ». Autrement dit une base homogène avec des briques toutes prêtes à disposition. Il existe des frameworks pour tous les langages de programmation et en particulier pour PHP. En faire la liste serait laborieux tant il en existe !

L'utilité d'un framework est d'éviter de passer du temps à développer ce qui a déjà été fait par d'autres souvent plus compétents et qui a en plus été utilisé et validé par de nombreux utilisateurs. On peut imaginer un framework comme un ensemble d'outils à disposition. Par exemple je dois faire du routage pour mon site, je prends un composant déjà tout prêt et qui a fait ses preuves et je l'utilise : gain de temps, fiabilité, mise à jour si nécessaire…

Il serait vraiment dommage de se passer d'un framework alors que le fait d'en utiliser un ne présente que des avantages.

I-B - Pourquoi Laravel ?

I-B-1 - Constitution de Laravel

Laravel, créé par Taylor Otwel, initie une nouvelle façon de concevoir un framework en utilisant ce qui existe de mieux pour chaque fonctionnalité. Par exemple toute application web a besoin d'un système qui gère les requêtes HTTP. Plutôt que de réinventer quelque chose, le concepteur de Laravel a tout simplement utilisé celui de **Symfony** en l'étendant pour créer un système de routage efficace. De la même manière, l'envoi des emails se fait avec la bibliothèque **SwiftMailer**. En quelque sorte Otwel a fait son marché parmi toutes les bibliothèques disponibles. Nous verrons dans ce cours comment cela est réalisé. Mais Laravel n'est pas seulement le regroupement de bibliothèques existantes, c'est aussi de nombreux composants originaux et surtout une orchestration de tout ça.

Vous allez trouver dans Laravel :

- un système de routage (RESTFul et ressources) ;
- un créateur de requêtes SQL et un ORM ;
- un moteur de template ;
- un système d'authentification pour les connexions ;
- un système de validation ;
- un système de pagination ;
- un système de migration pour les bases de données ;
- un système d'envoi d'emails ;
- un système de cache ;
- un système de gestion des événements ;
- un système d'autorisations ;
- une gestion des sessions ;
- un système de localisation ;
- un système de notifications.

Et d'autres choses encore que nous allons découvrir ensemble. Il est probable que certains éléments de cette liste ne vous évoquent pas grand-chose, mais ce n'est pas important pour le moment, tout cela deviendra plus clair au fil des chapitres.

I-B-2 - Le meilleur de PHP

Plonger dans le code de Laravel, c'est recevoir un cours de programmation tant le style est clair et élégant et le code bien organisé. La version actuelle de Laravel est la 5.5 (à la date de publication de ce tutoriel), elle nécessite au minimum la version 7 de PHP. Pour aborder de façon efficace ce framework, il serait souhaitable que vous soyez familiarisé avec ces notions :

- **les espaces de noms** : c'est une façon de bien ranger le code pour éviter des conflits de nommage. Laravel utilise cette possibilité de façon intensive. Tous les composants sont rangés dans des espaces de noms distincts, de même que l'application créée.
- **les fonctions anonymes** : ce sont des fonctions sans nom (souvent appelées *closures*) qui permettent d'améliorer le code. Les utilisateurs de JavaScript y sont habitués. Les utilisateurs de PHP un peu moins parce qu'elle y sont plus récentes. Laravel les utilise aussi de façon systématique.
- **les méthodes magiques** : ce sont des méthodes qui ne sont pas explicitement appelées dans le code PHP mais automatiquement appelées par l'interpréteur PHP en cas de besoin (l'exemple type est la méthode __tostring appelée pour convertir un objet en string).
- **les interfaces** : une interface est un contrat de constitution des classes. En programmation objet c'est le sommet de la hiérarchie. Tous les composants de Laravel sont fondés sur des interfaces.

- **les traits** : c'est une façon d'ajouter des propriétés et méthodes à une classe sans passer par l'héritage, ce qui permet de passer outre certaines limitations de l'héritage simple proposé par défaut par PHP.

Un framework n'est pas fait pour remplacer la connaissance d'un langage mais pour assister celui (ou celle) qui connaît déjà bien ce langage. Si vous avez des lacunes il vaut mieux les combler pour profiter pleinement de Laravel.

I-C - La documentation

Quand on s'intéresse à un framework il ne suffit pas qu'il soit riche et performant, il faut aussi que la documentation soit à la hauteur. C'est le cas pour Laravel. Vous trouverez la documentation **sur le site officiel**. Mais il existe de plus en plus de sources d'informations dont voici les principales :

- **https://laravel.fr** : site d'entraide francophone avec un forum actif ;
- **http://laravel.io** : le forum officiel ;
- **http://cheats.jesse-obrien.ca** : une page bien pratique qui résume toutes les commandes ;
- **http://www.laravel-tricks.com** : un autre site d'astuces ;
- **http://packalyst.com** : le rassemblement de tous les packages pour ajouter des fonctionnalités à Laravel ;
- **https://laracasts.com** : de nombreux tutoriels vidéo en anglais dont un certain nombre en accès gratuit ;
- **https://github.com/chiraggude/awesome-laravel** : une page comportant une multitude de liens intéressants.

Il existe aussi de bons livres mais pratiquement tous en anglais.

I-D - MVC ? POO ?

I-D-1 - MVC

On peut difficilement parler d'un framework sans évoquer le patron **Modèle-Vue-Contrôleur**. Pour certains il s'agit de la clé de voûte de toute application rigoureuse, pour d'autres c'est une contrainte qui empêche d'organiser judicieusement son code. De quoi s'agit-il ? Voici un petit schéma pour y voir clair :

C'est un modèle d'organisation du code :

- le modèle est chargé de gérer les données ;
- la vue est chargée de la mise en forme pour l'utilisateur ;
- le contrôleur est chargé de gérer l'ensemble.

En général on résume en disant que le modèle gère la base de données, la vue produit les pages HTML et le contrôleur fait tout le reste. Dans Laravel :

- le modèle correspond à une table d'une base de données. C'est une classe qui étend la classe **Model** qui permet une gestion simple et efficace des manipulations de données et l'établissement automatisé de relations entre tables ;
- le contrôleur se décline en deux catégories : contrôleur classique et contrôleur de ressource (je détaillerai évidemment tout ça dans le cours) ;
- la vue est soit un simple fichier avec du code HTML, soit un fichier utilisant le système de template **Blade** de Laravel.

Laravel propose ce patron mais ne l'impose pas. Nous verrons d'ailleurs qu'il est parfois judicieux de s'en éloigner parce qu'il y a des tas de chose qu'on n'arrive pas à caser dans cette organisation. Par exemple si je dois envoyer des

emails où vais-je placer mon code ? En général ce qui se produit est l'inflation des contrôleurs auxquels on demande des choses pour lesquelles ils ne sont pas faits.

I-D-2 - POO

Laravel est fondamentalement orienté objet. La POO est un modèle de programmation qui s'éloigne radicalement de la programmation procédurale. Avec la POO tout le code est placé dans des classes qui découlent d'interfaces qui établissent des contrats de fonctionnement. Avec la POO on manipule des objets.

Avec la POO, la responsabilité du fonctionnement est répartie dans des classes, alors que dans l'approche procédurale tout est mélangé. Le fait de répartir la responsabilité évite la duplication du code qui est le lot presque forcé de la programmation procédurale. Laravel pousse au maximum cette répartition en utilisant l'injection de dépendance.

L'utilisation de classes bien identifiées, dont chacune a un rôle précis, pilotées par des interfaces claires, dopées par l'injection de dépendances : tout cela crée un code élégant, efficace, lisible, facile à maintenir et à tester. C'est ce que Laravel propose. Alors vous pouvez évidemment greffer là-dessus votre code approximatif, mais vous pouvez aussi vous inspirer des sources du framework pour améliorer votre style de programmation.

L'injection de dépendance est destinée à éviter de rendre les classes dépendantes et à privilégier une liaison dynamique plutôt que statique. Le résultat est un code plus lisible, plus facile à maintenir et à tester. Nous verrons ce mécanisme à l'œuvre dans Laravel.

I-E - En résumé

- Un framework fait gagner du temps et donne l'assurance de disposer de composants bien codés et fiables.
- Laravel est un framework novateur, complet, qui utilise les possibilités les plus récentes de PHP et qui est impeccablement codé et organisé.
- La documentation de Laravel est complète, précise et de nombreux tutoriels et exemples sont disponibles sur la toile.
- Laravel adopte le patron MVC mais ne l'impose pas, il est totalement orienté objet.

II - Un environnement de développement

Pour fonctionner Laravel a besoin d'un certain environnement :

- un serveur ;
- PHP ;
- MySQL ;
- Node ;
- Composer.

Il y a de plus en plus d'environnements disponibles dans le cloud avec des options gratuites comme chez **c9**. Mais rien ne vaut un bon système local : c'est rapide, sûr et on peut tout gérer. Mais évidemment il faut se le construire !

Heureusement il existe des solutions toutes prêtes, par exemple pour PHP + MySQL : **wampserver**, **xampp**, **easyphp**…

Ces solutions sont intéressantes mais pour ce cours je vous conseille plutôt **Laragon**. Je l'apprécie de plus en plus en abandonnant peu à peu WAMP que j'utilise depuis des années. Il est simple, rapide, convivial, non intrusif, complet, et en plus pensé pour Laravel ! Mais il ne fonctionne que sur Windows.

Pour les utilisateurs de Linux il faut se tourner vers l'une des solutions évoquées ci-dessus ou alors Homestead qui est l'environnement officiel de Laravel. Pour son installation il suffit de suivre la procédure décrite dans la documentation.

Par contre je déconseille aux utilisateurs de Windows d'installer **Homestead**, sauf s'ils ont envie de passer de longues heures à configurer leur système.

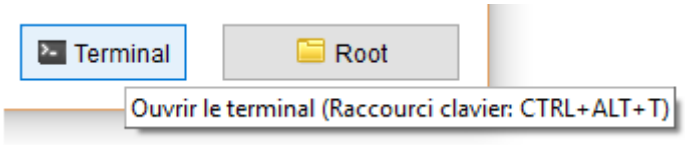
Donc en résumé :

- pour Windows : Laragon ;
- pour Linux ou Mac : Homestead.

II-A - Laragon

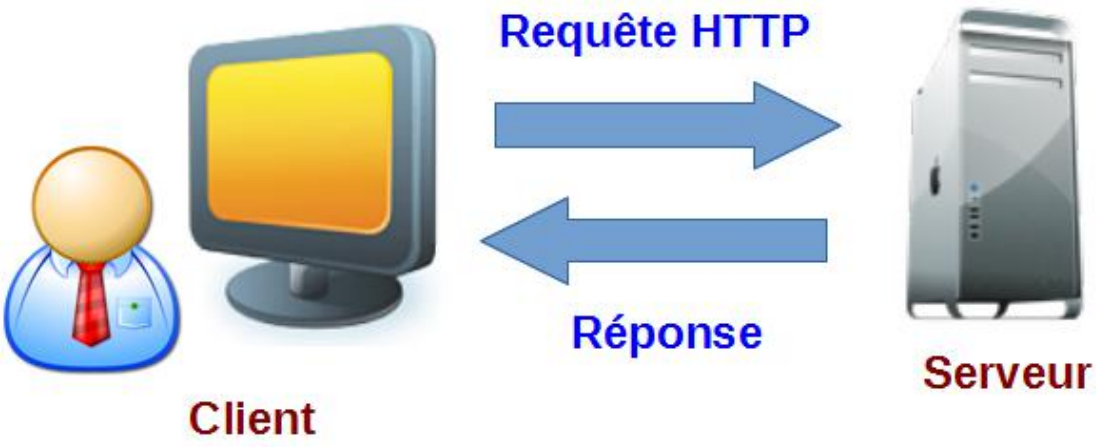
II-A-1 - Installation

Pour récupérer Laragon il faut se rendre sur le site :



Il n'y a que cette page, ne cherchez pas trop de documentation (mais ne vous en faites pas, il n'y en a pas vraiment besoin) ; il y a surtout un bouton pour aller sur le forum de discussion ainsi qu'un autre pour le téléchargement. Vous verrez que vous avez plusieurs possibilités pour le téléchargement, choisissez celle qui vous convient, a priori WAMP.

Une fois l'installation effectuée vous ouvrez Laragon :



Personnellement j'utilise encore la version 2.2.2 qui correspond tout à fait à mes besoins.

Vous n'avez plus qu'à cliquer sur « Tout démarrer » pour créer et lancer le serveur :

Status	Method	File	Domain	Cause	Type
200	GET	/	laravel.com	document	html
200	GET	css?family=Miriam+Libre:400,700|Source+Sans...	fonts.googleapis.com	stylesheet	css
200	GET	laravel-c76147199f.css	laravel.com	stylesheet	css
200	GET	flexboxgrid.min.css	cdnjs.cloudflare.com	stylesheet	css
200	GET	laravel-logo-white.png	laravel.com	img	png
200	GET	vue.min.js	cdnjs.cloudflare.com	script	js
200	GET	ui-preview.png	forge.laravel.com	img	png
200	GET	laravel-7310ca5785.js	laravel.com	script	js
200	GET	viewport-units-buggyfill.js	laravel.com	script	js
200	GET	cloud-bar.png	laravel.com	img	png
200	GET	laravel-tucked.png	laravel.com	img	png
200	GET	forge-flames.jpg	laravel.com	img	jpeg
200	GET	Ljtpu8zR5iJWmIN3Faba5egdm0LZdjqr5-oayXSO...	fonts.gstatic.com	font	woff2
200	GET	FLc0J-Gdn8ynDWUkeeesAHNuWYKPzoeKI5tYj8...	fonts.gstatic.com	font	woff2
200	GET	ga.js	ssl.google-analytics.com	script	js
200	GET	laravel-c76147199f.css	laravel.com	stylesheet	css
200	GET	__utm.gif?utmwv=5.6.7&utms=1&utmn=46500...	ssl.google-analytics.com	img	gif

Vous avez maintenant à disposition :

- un serveur Apache ;
- **PHP 7** ;
- **MySQL 5.7** (**MariaDB**) ;
- l'éditeur **Notepad++** ;
- le terminal **Cmder** (une console améliorée par rapport à l'horrible de Windows !) ;
- **composer** ;
- **nodejs** ;
- **putty** ;
- **memcached** ;
- **git** ;
- **redis**.

Avec Homestead on obtient pratiquement les mêmes possibilités.

Ne vous inquiétez pas si vous ne connaissez pas la moitié de ces outils ! D'une part nous ne les utiliserons pas tous, d'autre part je détaillerai l'utilisation de ceux qui vous seront nécessaires.

II-A-2 - Hôte virtuel

Cerise sur le gâteau : Laragon crée automatiquement des hôtes virtuels pour les dossiers qui se trouvent sur le serveur (**www**). Pour ceux qui ne savent pas de quoi il s'agit voici un exemple avec justement le cas de Laravel. Le fichier de démarrage de Laravel est placé en **www/monsite/public/index.html**. Donc à partir de **localhost** il faut entrer : **localhost/monsite/public**. Ce n'est ni pratique ni élégant. Un hôte virtuel permet de définir une adresse simplifiée. Par exemple ici Laragon va définir automatiquement **monsite.dev**. Avouez que c'est quand même mieux !

Mais ce n'est pas seulement une histoire d'esthétique ou d'économie de clavier. Le fait de disposer d'un hôte virtuel permet d'avoir en local exactement le même comportement que sur le serveur de production. Par exemple si vous avez sur une page HTML une image avec ce genre de référence : **/images/bouton.png**, vous serez tout à fait heureux d'avoir un hôte virtuel pour que l'image s'affiche !

Créer manuellement un hôte virtuel avec Windows n'est pas difficile mais un peu laborieux. Il faut modifier le fichier **hosts** de Windows et **httpd-vhosts.conf** de Apache. Autant laisser Laragon s'en charger !

II-B - Composer

II-B-1 - Présentation

Je vous ai dit dans le précédent chapitre que Laravel utilise des composants d'autres sources. Plutôt que de les incorporer directement, il utilise un gestionnaire de dépendances : **composer**. D'ailleurs pour le coup les composants de Laravel sont aussi traités comme des dépendances. Mais c'est quoi un gestionnaire de dépendances ?

Imaginez que vous créez une application PHP et que vous utilisez des composants issus de différentes sources : **Carbon** pour les dates, **Redis** pour les données… Vous pouvez utiliser la méthode laborieuse en allant chercher tout ça de façon manuelle, et vous allez être confronté à des difficultés :

- télécharger tous les composants dont vous avez besoin et les placer dans votre structure de dossiers ;
- traquer les éventuels conflits de nommage entre les librairies ;
- mettre à jour manuellement les librairies quand c'est nécessaire ;
- prévoir le code pour charger les classes à utiliser.

Tout ça est évidemment faisable mais avouez que s'il était possible d'automatiser les procédures ce serait vraiment génial. C'est justement ce que fait un gestionnaire de dépendances !

II-C - JSON

Pour comprendre le fonctionnement de composer, il faut connaître le format **JSON** qui est l'acronyme de **JavaScript Object Notation**. Un fichier JSON a pour but de contenir des informations de type étiquette-valeur. Regardez cet exemple élémentaire :

```
1. {
2.   "nom": "Durand",
3.   "prénom": "Jean"
4. }
```

Les étiquettes sont « nom » et « prénom » et les valeurs correspondantes « Durand » et « Jean ». Les valeurs peuvent être aussi des tableaux ou des objets. Regardez ce second exemple :

```
1. {
2.     "identité1" : {
3.       "nom": "Durand",
4.       "prénom": "Jean"
5.     },
6.     "identité2" : {
7.       "nom": "Dupont",
8.       "prénom": "Albert"
9.     }
10. }
```

Composer a besoin d'un fichier **composer.json** associé. Ce fichier contient les instructions pour composer : les dépendances, les classes à charger automatiquement… Par exemple :

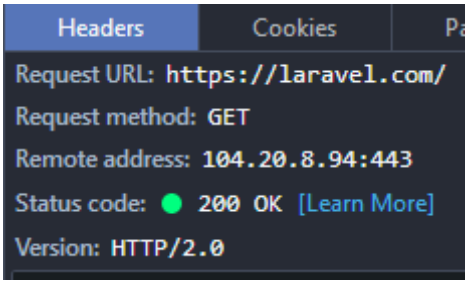
```
1. "require": {
2.     "php": ">=7.0.0",
3.     "laravel/framework": "5.5.*",
4. },
```

Ici on dit qu'on veut (**require**) que PHP soit au moins en version 7.0.0 et on veut également charger le composant « laravel/framework ».

On verra comment se présente le fichier **composer.json** de Laravel dans le prochain chapitre.

II-C-1 - Packagist

Tous les composants disponibles se trouvent sur le site **Packagist** :

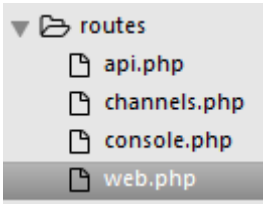


Par exemple il me faut un composant pour l'envoi d'email, j'entre ceci dans la zone de recherche :

J'obtiens une liste assez longue et je n'ai plus qu'à fouiller un peu pour trouver ce que je cherche.

II-C-2 - Packalyst

Le site **Packalyst** est spécialisé dans les composants conçus pour Laravel :

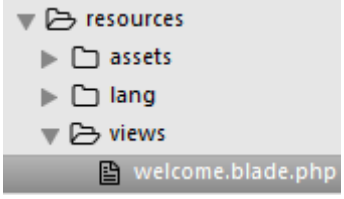


Là vous êtes sûr que le composant va fonctionner directement dans Laravel ! Il faut toutefois vérifier qu'il correspond au numéro de version que vous utilisez.

Pour aller plus loin avec composer vous pouvez lire **cet article**.

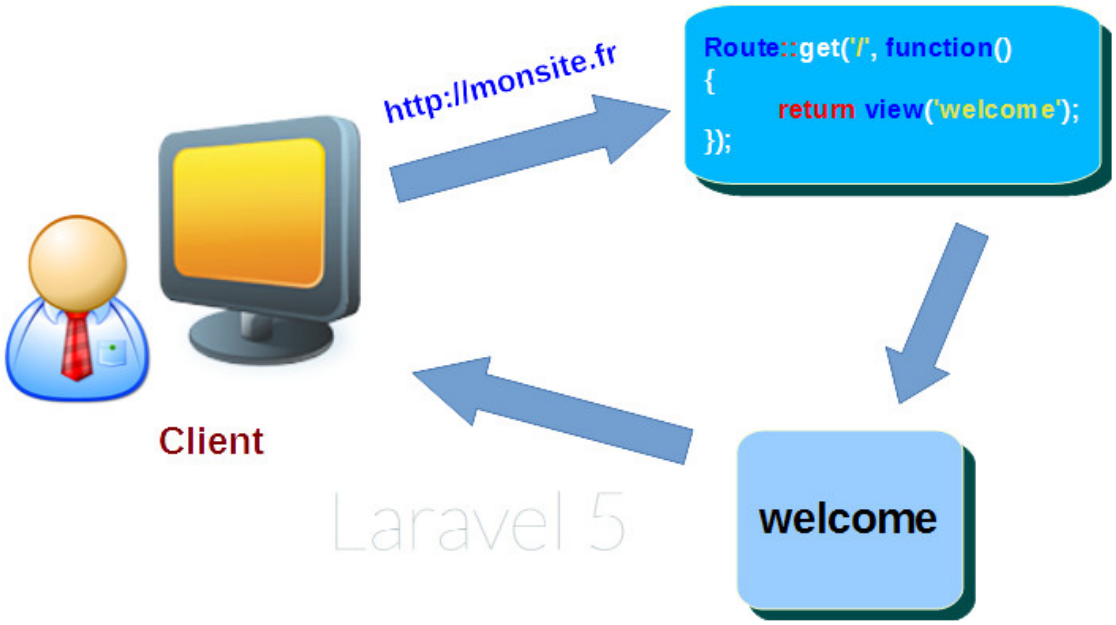
II-D - Les éditeurs de code

Choisir un éditeur de code n'est pas évident, les critères sont nombreux. J'aime bien **Sublime Text** qui jouit d'une grande popularité :

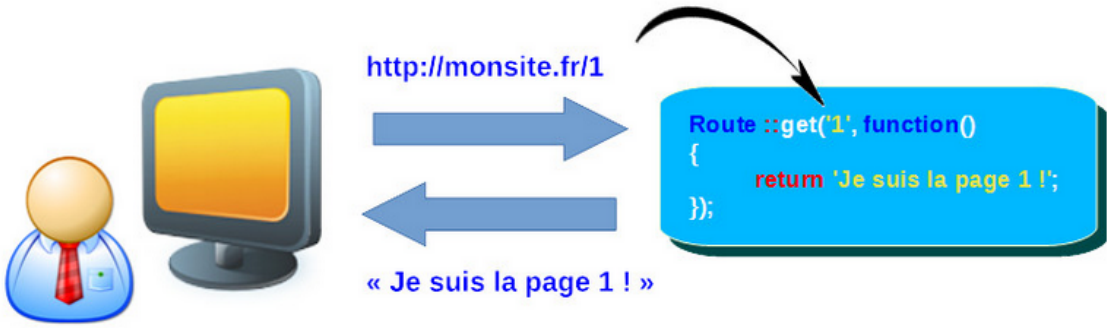


Son grand avantage est de proposer de très nombreux plugins pour étendre ses possibilités et il y en a pas mal pour Laravel. Mais pour ça il faut commencer par installer le **Package Control**. Vous trouvez la procédure d'installation **ici**.

Ensuite vous pouvez installer tous les plugins que vous voulez. Pour trouver ceux qui concernent Laravel vous avez une zone de recherche :



Dans la communauté Laravel, l'IDE qui a le plus de succès est **PhpStorm**. Il est vraiment puissant et complet mais il n'est pas gratuit :



II-E - En résumé

- Laravel a besoin d'un environnement de développement complet : PHP, MySQL, composer…
- Il existe des solutions toutes prêtes : Homestead pour Linux et Laragon pour Windows.
- Composer est le gestionnaire de dépendances utilisé par Laravel.
- Sublime Text est l'éditeur de code le plus utilisé.
- L'IDE préféré des utilisateurs de Laravel est PhpStorm.

III - Installation et organisation

Dans ce chapitre nous allons voir comment créer une application Laravel et comment le code est organisé dans une application.

Pour utiliser Laravel et suivre ce chapitre et l'ensemble du cours vous aurez besoin d'un serveur équipé de PHP avec au minimum la version 7 et aussi de MySQL. Nous avons vu dans le précédent chapitre les différentes possibilités.

Quelle que soit l'application que vous utilisez, vérifiez que vous avez la bonne version de PHP (minimum 7). D'autre part plusieurs extensions de PHP doivent être activées.

III-A - Créer une application Laravel

III-A-1 - Le serveur

Pour fonctionner correctement, Laravel a besoin de PHP :

- version >= 7.0.0 ;
- extension PDO ;
- extension Mbstring ;
- extension OpenSSL ;
- extension Tokenizer ;
- extension XML.

Laravel est équipé d'un serveur sommaire pour le développement qui se lance avec cette commande :

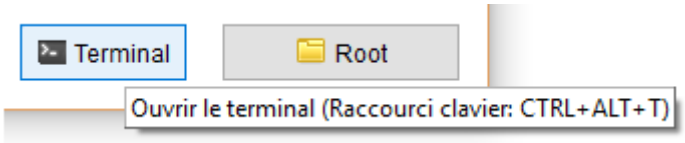
```
php artisan serve
```

On y accède à cette adresse : **http://localhost:8000**. Mais évidemment pour que ça fonctionne il faut que vous ayez PHP installé.

III-A-2 - Prérequis

Composer fonctionne en ligne de commande. Vous avez donc besoin de la console (nommée Terminal ou Konsole sur OS X et Linux). Les utilisateurs de Linux sont très certainement habitués à l'utilisation de la console mais il n'en est généralement pas de même pour les adeptes de Windows. Pour trouver la console sur ce système il faut chercher l'invite de commande :

Si vous utilisez Laragon, comme je vous le conseille, vous avez une console améliorée (**Cmder**) accessible avec ce bouton.

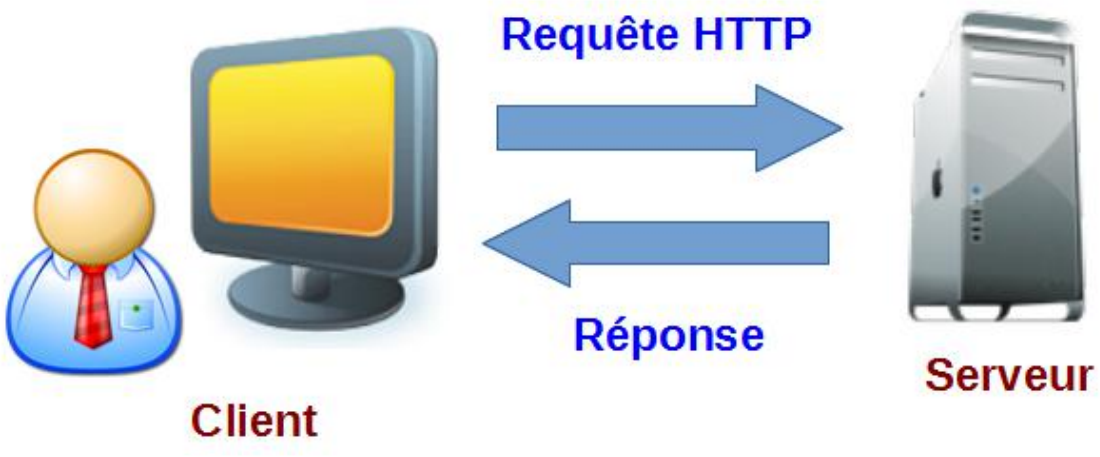


III-A-3 - Installation avec Composer

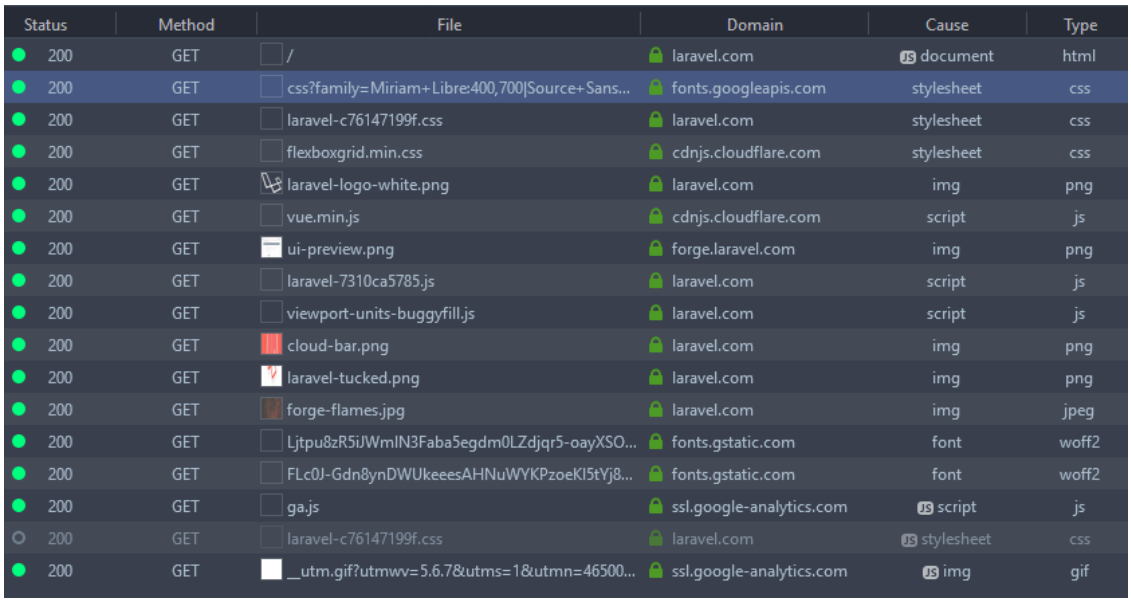
Il y a plusieurs façons de créer une application Laravel. La plus classique consiste à utiliser la commande **create-project** de composer. Par exemple je veux créer une application dans un dossier laravel5 à la racine de mon serveur, voici la syntaxe à utiliser :

```
composer create-project --prefer-dist laravel/laravel laravel5
```

L'installation démarre et je n'ai plus qu'à attendre quelques minutes pour que composer fasse son travail jusqu'au bout. Vous verrez s'afficher une liste de téléchargements. Finalement on se retrouve avec cette architecture :



On peut vérifier que tout fonctionne bien avec l'URL **http://localhost/laravel5/public**. Normalement on doit obtenir cette page très épurée :



Status	Method	File	Domain	Cause	Type
200	GET	/	laravel.com	document	html
200	GET	css?family=Miriam+Libre:400,700|Source+Sans...	fonts.googleapis.com	stylesheet	css
200	GET	laravel-c76147199f.css	laravel.com	stylesheet	css
200	GET	flexboxgrid.min.css	cdnjs.cloudflare.com	stylesheet	css
200	GET	laravel-logo-white.png	laravel.com	img	png
200	GET	vue.min.js	cdnjs.cloudflare.com	script	js
200	GET	ui-preview.png	forge.laravel.com	img	png
200	GET	laravel-7310ca5785.js	laravel.com	script	js
200	GET	viewport-units-buggyfill.js	laravel.com	script	js
200	GET	cloud-bar.png	laravel.com	img	png
200	GET	laravel-tucked.png	laravel.com	img	png
200	GET	forge-flames.jpg	laravel.com	img	jpeg
200	GET	Ljtpu8zR5iJWmIN3Faba5egdm0LZdjqr5-oayXSO...	fonts.gstatic.com	font	woff2
200	GET	FLc0J-Gdn8ynDWUkeeesAHNuWYKPzoeKI5tYj8...	fonts.gstatic.com	font	woff2
200	GET	ga.js	ssl.google-analytics.com	script	js
200	GET	laravel-c76147199f.css	laravel.com	stylesheet	css
200	GET	__utm.gif?utmwv=5.6.7&utms=1&utmn=46500...	ssl.google-analytics.com	img	gif

Pour les mises à jour ultérieures il suffit encore d'utiliser composer avec la commande **update** :

```
composer update
```

III-A-4 - Installation avec Laravel Installer

Une autre solution pour installer Laravel consiste à utiliser l'installeur. Il faut commencer par installer globalement l'installeur avec composer :

```
composer global require "laravel/installer"
```

Il faut ensuite informer la variable d'environnement **path** de l'emplacement du dossier **…/composer/vendor/bin**.

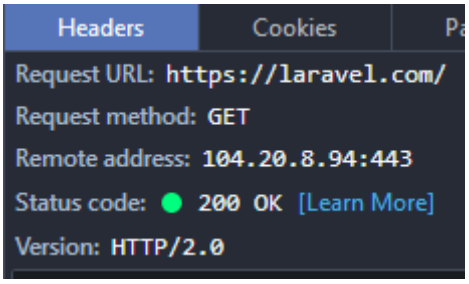
Pour créer une application il suffit de taper :

```
laravel new monAppli
```

Laravel sera alors installé dans le dossier **monAppli**.

III-A-5 - Installation avec Laragon

Une façon encore plus simple pour installer Laravel est d'utiliser Laragon avec le menu :



On vous demande alors le nom du projet (donc du dossier) :

Et ensuite vous n'avez plus qu'à attendre ! La console s'ouvre et vous pouvez suivre le déroulement des opérations. On vous rappelle la commande de composer et en plus une base de données est automatiquement créée avec le même nom !

*Si vous installez Laravel en téléchargeant directement les fichiers sur Github et en utilisant la commande **composer install** il vous faut effectuer deux actions complémentaires. En effet dans ce cas il ne sera pas automatiquement créé de clé de sécurité et vous allez tomber sur une erreur au lancement. Il faut donc la créer avec la commande **php artisan key:generate**. D'autre part vous aurez à la racine le fichier **.env.example** que vous devrez renommer en **.env** (ou en faire une copie) pour que la configuration fonctionne.*

III-A-6 - Autorisations

Au niveau des dossiers de Laravel, les seuls qui ont besoin de droits d'écriture par le serveur sont **storage** (et ses sous-dossiers), et **bootstrap/cache**.

III-B - Des URL propres

Pour un serveur Apache il est prévu dans le dossier **public** un fichier **.htaccess** avec ce code :

```
1. <IfModule mod_rewrite.c>
2.     <IfModule mod_negotiation.c>
3.         Options -MultiViews
4.     </IfModule>
5.
6.     RewriteEngine On
7.
8.     # Redirect Trailing Slashes If Not A Folder...
9.     RewriteCond %{REQUEST_FILENAME} !-d
10.     RewriteCond %{REQUEST_URI} (.+)/$
11.     RewriteRule ^ %1 [L,R=301]
12.
```

```
    # Handle Front Controller...
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]

    # Handle Authorization Header
    RewriteCond %{HTTP:Authorization} .
    RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
</IfModule>
```

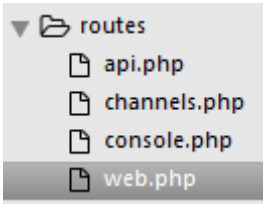
Le but est essentiellement d'éviter d'avoir **index.php** dans l'URL. Mais pour que ça fonctionne il faut activer le module **mod_rewrite**.

III-C - Organisation de Laravel

Maintenant qu'on a un Laravel tout neuf et qui fonctionne voyons un peu ce qu'il contient…

III-C-1 - Dossier app

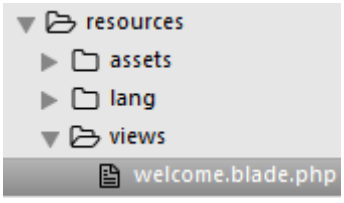
Ce dossier contient les éléments essentiels de l'application :



- **Console** : toutes les commandes en mode console ;
- **Http** : tout ce qui concerne la communication : contrôleurs, middlewares (il y a 4 middlewares de base qui servent à filtrer les requêtes HTTP) et le kernel ;
- **Providers** : tous les fournisseurs de services (providers), il y en a déjà 5 au départ. Les providers servent à initialiser les composants ;
- **User** : un modèle qui concerne les utilisateurs pour la base de données.

Évidemment tout cela doit vous paraître assez nébuleux pour le moment mais nous verrons en détail ces éléments au fil du cours. Et on verra d'ailleurs que seront créés bien d'autres dossiers selon nos besoins.

III-C-2 - Autres dossiers

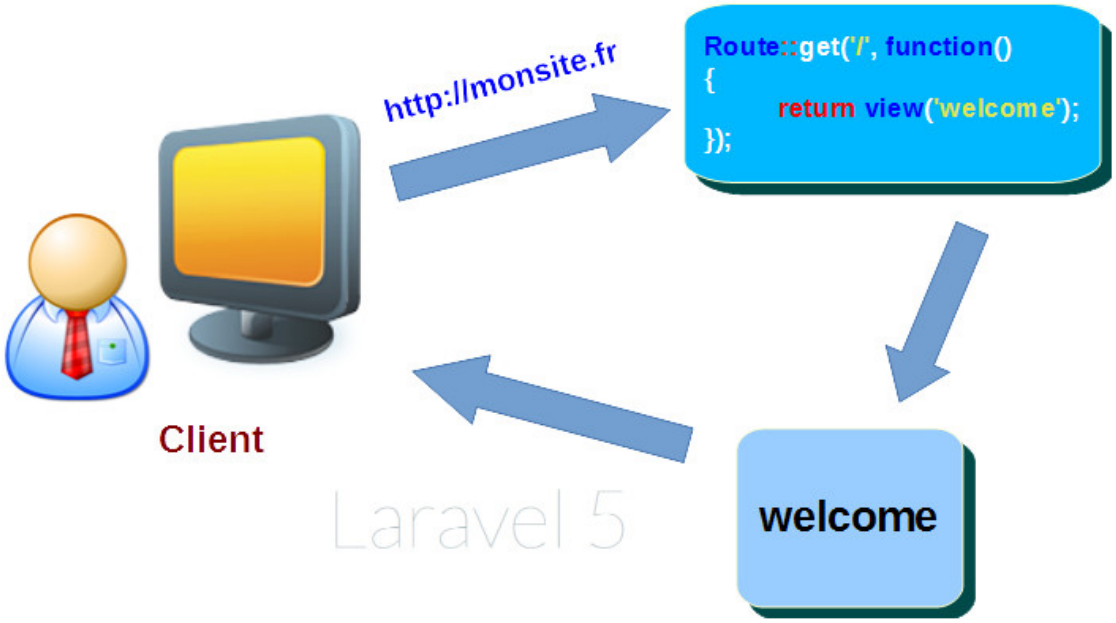


Voici une description du contenu des autres dossiers :

- **bootstrap** : scripts d'initialisation de Laravel pour le chargement automatique des classes, la fixation de l'environnement et des chemins et pour le démarrage de l'application ;
- **public** : tout ce qui doit apparaître dans le dossier public du site : images, CSS, scripts… ;
- **config** : toutes les configurations : application, authentification, cache, base de données, espaces de noms, emails, systèmes de fichier, session… ;
- **database** : migrations et populations ;
- **resources** : vues, fichiers de langage et assets (par exemple les fichiers Sass) ;

- **routes** : la gestion des URL d'entrée de l'application ;
- **storage** : données temporaires de l'application : vues compilées, caches, clés de session… ;
- **tests** : fichiers de tests ;
- **vendor** : tous les composants de Laravel et de ses dépendances (créé par composer).

III-C-3 - Fichiers de la racine



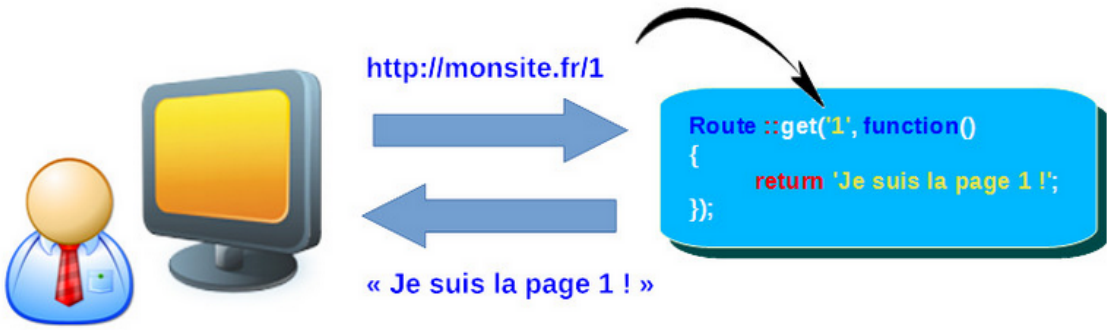
Il y a un certain nombre de fichiers dans la racine dont voici les principaux :

- **artisan** : outil en ligne de Laravel pour des tâches de gestion ;
- **composer.json** : fichier de référence de composer ;
- **package.json** : fichier de référence de npm pour les assets ;
- **phpunit.xml** : fichier de configuration de phpunit (pour les tests unitaires) ;
- **.env** : fichier pour spécifier l'environnement d'exécution.

Nous verrons tout cela progressivement dans le cours, ne vous inquiétez pas !

III-C-4 - Accessibilité

Pour des raisons de sécurité sur le serveur, seul le dossier **public** doit être accessible :



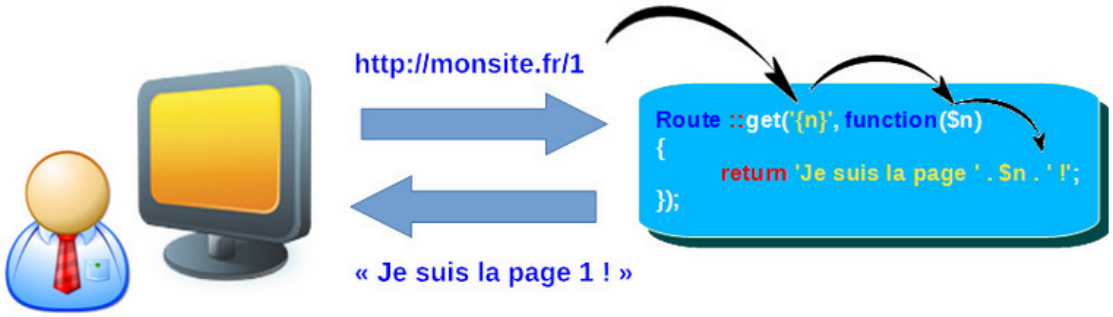
Cette configuration n'est pas toujours possible sur un serveur mutualisé, il faut alors modifier un peu Laravel pour que ça fonctionne ; j'en parlerai dans le chapitre sur le déploiement.

III-D - Environnement et messages d'erreur

Par défaut lorsque vous installez Laravel, celui-ci est en mode « debug » et vous aurez une description précise de toutes les erreurs. Par exemple ouvrez le fichier **routes/web.php** et changez ainsi le code :

```
Route::post('/', function () {
    return view('welcome');
});
```

Ouvrez l'URL de base, vous obtenez cette page :



Laravel utilise la librairie **Whoops** *pour les erreurs.*

Pendant la phase de développement on a besoin d'obtenir des messages explicites pour traquer les erreurs inévitables que nous allons faire. En mode « production » il faudra changer ce mode, pour cela ouvrez le fichier **.env** et trouvez cette ligne :

```
'debug' => env('APP_DEBUG', false),
```

Autrement dit on va chercher la valeur dans l'environnement, mais où peut-on le trouver ? Regardez à la racine des dossiers, vous y trouvez le fichier **.env**.

Avec ce contenu :

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:yrTZiYq/TxheZ3tBcqHo4X47uuSXROCY5gg3H0Pljt8=
APP_DEBUG=true
APP_LOG_LEVEL=debug
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

BROADCAST_DRIVER=log
CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
```

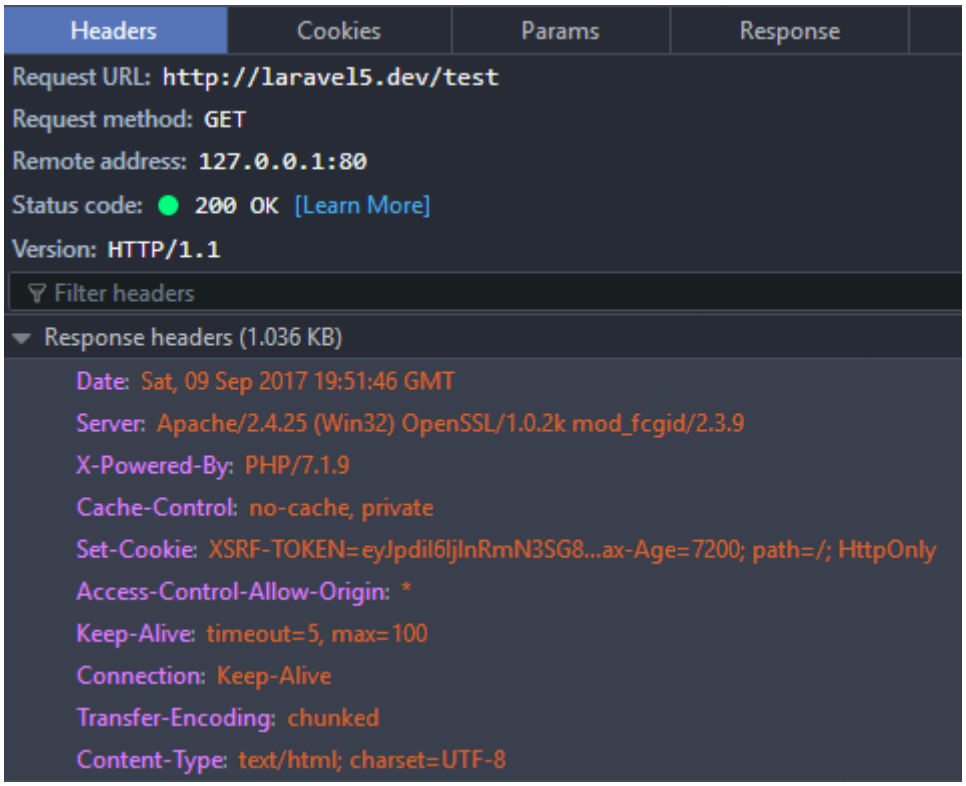
```
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
```

Vous remarquez que dans ce fichier la variable **APP_DEBUG** a la valeur **true**. On va la conserver ainsi puisqu'on veut être en mode « debug ». Vous êtes ainsi en mode débogage avec affichage de messages d'erreur détaillés. Si vous la mettez à **false** (ou si vous la supprimez), avec une URL non prévue vous obtenez maintenant juste :

Sorry, the page you are looking for could not be found.

Il ne faudra évidemment pas laisser la valeur **true** lors d'une mise en production ! On en reparlera lorsqu'on verra la gestion de l'environnement. Vous ne risquerez ainsi plus d'oublier de changer cette valeur parce que Laravel saura si vous êtes sur votre serveur de développement ou sur celui de production.

D'autre part il y a un fichier qui collecte les erreurs (Log), vous le trouvez ici :



Il n'existe pas à l'installation mais se crée à la première erreur signalée.

Par défaut il n'y a qu'un fichier mais si vous préférez avoir un fichier par jour par exemple il suffit d'ajouter cette ligne dans le fichier **.env** :

```
APP_LOG=daily
```

De la même manière par défaut Laravel stocke toutes les erreurs. C'est pratique dans la phase de développement mais en production vous pouvez limiter le niveau de sévérité des erreurs retenues, par exemple si vous vous contentez des warning :

```
APP_LOG_LEVEL=warning
```

Laravel utilise le composant **Monolog** pour la gestion des erreurs. Reportez-vous à sa documentation si vous avez besoin de plus d'informations. **La documentation de Laravel** en résume l'essentiel ainsi que son adaptation au framework.

*La valeur de **APP_KEY** qui sécurise les informations est automatiquement générée lors de l'installation avec **create-project**.*

III-E - Une application d'exemple

Une application d'exemple complète servira de fil conducteur pour ce cours essentiellement à partir de sa deuxième partie mais vous pouvez déjà l'installer et la parcourir. Vous avez toutes les informations pour cette installation **ici**.

Plutôt que de créer des petits morceaux d'application limités et pas réalistes je me référerai à cette application complète pour certains chapitres de ce cours, comme par exemple tout ce qui concerne la gestion des bases de données.

Une grande partie du code de cette application d'exemple ne sera pas traitée dans ce cours et fera sans doute l'objet d'articles séparés. D'autre part cet exemple sera évolutif et suivra les suggestions que je reçois sur Github. Je le tiendrai également à jour en fonction des PR (Pull Request de Gihub) du projet Laravel.

III-F - En résumé

- Pour son installation et sa mise à jour Laravel utilise le gestionnaire de dépendances **composer.**
- La création d'une application Laravel se fait à partir de la console avec une simple ligne de commande.
- Laravel est organisé en plusieurs dossiers.
- Le dossier **public** est le seul qui doit être accessible pour le client.
- L'environnement est fixé à l'aide du fichier **.env.**
- Par défaut Laravel est en mode « debug » avec affichage de toutes les erreurs.

IV - Le routage

Dans ce chapitre nous allons nous intéresser au devenir d'une requête HTTP qui arrive dans notre application Laravel. Nous allons voir l'intérêt d'utiliser un fichier **.htaccess** pour simplifier les URL. Nous verrons aussi le système de routage pour trier les requêtes.

IV-A - Les requêtes HTTP

IV-A-1 - Petit rappels

On va commencer par un petit rappel sur ce qu'est une requête HTTP. Voici un schéma illustratif :

Le HTTP (Hypertext Transfer Protocol) est un protocole de communication entre un client et un serveur. Le client demande une page au serveur en envoyant une requête et le serveur réagit en envoyant une réponse, en général une page HTML.

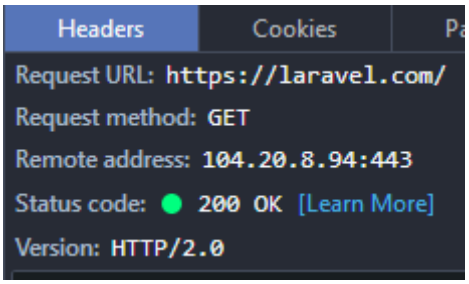
Quand on surfe sur Internet chacun de nos clics provoque en général cet échange, et plus généralement une rafale d'échanges.

La requête du client comporte un certain nombre d'informations (headers, status code, body…).

Prenons un exemple avec **le site de Laravel**. Lorsque je clique sur le lien, voici les requêtes HTTP qui se produisent :

Status	Method	File	Domain	Cause	Type
200	GET	/	laravel.com	document	html
200	GET	css?family=Miriam+Libre:400,700|Source+Sans...	fonts.googleapis.com	stylesheet	css
200	GET	laravel-c76147199f.css	laravel.com	stylesheet	css
200	GET	flexboxgrid.min.css	cdnjs.cloudflare.com	stylesheet	css
200	GET	laravel-logo-white.png	laravel.com	img	png
200	GET	vue.min.js	cdnjs.cloudflare.com	script	js
200	GET	ui-preview.png	forge.laravel.com	img	png
200	GET	laravel-7310ca5785.js	laravel.com	script	js
200	GET	viewport-units-buggyfill.js	laravel.com	script	js
200	GET	cloud-bar.png	laravel.com	img	png
200	GET	laravel-tucked.png	laravel.com	img	png
200	GET	forge-flames.jpg	laravel.com	img	jpeg
200	GET	Ljtpu8zR5iJWmIN3Faba5egdm0LZdjqr5-oayXSO...	fonts.gstatic.com	font	woff2
200	GET	FLc0J-Gdn8ynDWUkeeesAHNuWYKPzoeKl5tYj8...	fonts.gstatic.com	font	woff2
200	GET	ga.js	ssl.google-analytics.com	script	js
200	GET	laravel-c76147199f.css	laravel.com	stylesheet	css
200	GET	__utm.gif?utmwv=5.6.7&utms=1&utmn=46500...	ssl.google-analytics.com	img	gif

En tout 17 requêtes avec la méthode GET. Regardons d'un peu plus près la première :



On trouve :

- l'URL : **https://laravel.com/**
- la méthode : GET
- l'adresse IP : 104.20.8.94:443
- le code : 200 (donc tout s'est bien passé)
- la version du HTTP : 2.0

Il y a évidemment bien d'autres choses dans les headers (content-type, cookies, encodage…) mais pour le moment on va se contenter de ces informations. Notre navigateur digère tout ça de façon transparente, heureusement pour nous !

Notre application Laravel doit savoir interpréter les informations qui arrivent et les utiliser de façon pertinente pour renvoyer ce que demande le client. Nous allons voir comment cela est réalisé.

Il est souvent utile de générer des requêtes HTTP à partir d'un client, il existe de nombreux outils pour cela. Personnellement j'utilise le module de Firefox **HttpRequester**.

IV-A-2 - Les méthodes

Il est indispensable de connaître les principales méthodes du HTTP :

- **GET** : c'est la plus courante, on demande une ressource qui ne change jamais, on peut mémoriser la requête, on est sûr d'obtenir toujours la même ressource ;
- **POST** : elle est aussi très courante, la requête modifie ou ajoute une ressource, le cas le plus classique est la soumission d'un formulaire (souvent utilisé à tort à la place de PUT) ;
- **PUT** : on ajoute ou remplace complètement une ressource ;
- **PATCH** : on modifie partiellement une ressource (donc à ne pas confondre avec PUT) ;
- **DELETE** : on supprime une ressource.

La différence entre PUT et POST est loin d'être évidente, vous pouvez lire sur le sujet **cet excellent article**.

IV-A-3 - .htaccess et index.php

Pour Laravel on veut que toutes les requêtes aboutissent obligatoirement sur le fichier **index.php** situé dans le dossier **public**. Pour y arriver on peut utiliser une URL de ce genre :

```
http://monsite.fr/index.php/mapage
```

Mais ce n'est pas très esthétique avec ce **index.php** au milieu. Si vous avez un serveur Apache, lorsque la requête du client arrive sur le serveur où se trouve notre application Laravel, elle passe en premier par le fichier **.htaccess**, s'il existe, qui fixe des règles pour le serveur. Il y a justement un fichier **.htaccess** dans le dossier **public** de Laravel avec une règle de réécriture de telle sorte qu'on peut avoir une URL simplifiée :

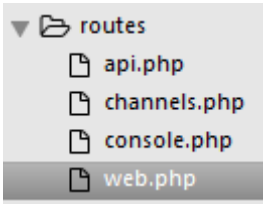
```
http://monsite.fr/mapage
```

Un petit schéma pour visualiser cette action :

*Pour que ça fonctionne il faut que le serveur Apache ait le module **mod_rewrite** activé.*

IV-A-4 - Le cycle de la requête

Lorsque la requête atteint le fichier **public/index.php** l'application Laravel est créée et configurée et l'environnement est détecté. Nous reviendrons plus tard plus en détail sur ces étapes. Ensuite le fichier **routes/web.php** est chargé. Voici l'emplacement de ce fichier :



*Les autres fichiers concernent des routes plus spécifiques comme pour les API avec le fichier **api.php** ou les routes pour les actions en ligne de commande avec le fichier **console.php**.*

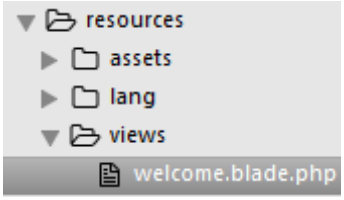
C'est avec ce fichier que la requête va être analysée et dirigée. Regardons ce qu'on y trouve au départ :

```
Route::get('/', function () {
    return view('welcome');
});
```

Comme Laravel est explicite vous pouvez déjà deviner à quoi sert ce code :

- **Route** : on utilise le routeur ;
- **get** : on regarde si la requête a la méthode « get » ;
- **'/'** : on regarde si l'URL comporte uniquement le nom de domaine ;
- dans la fonction anonyme on retourne (**return**) une vue (**view**) à partir du fichier « welcome ».

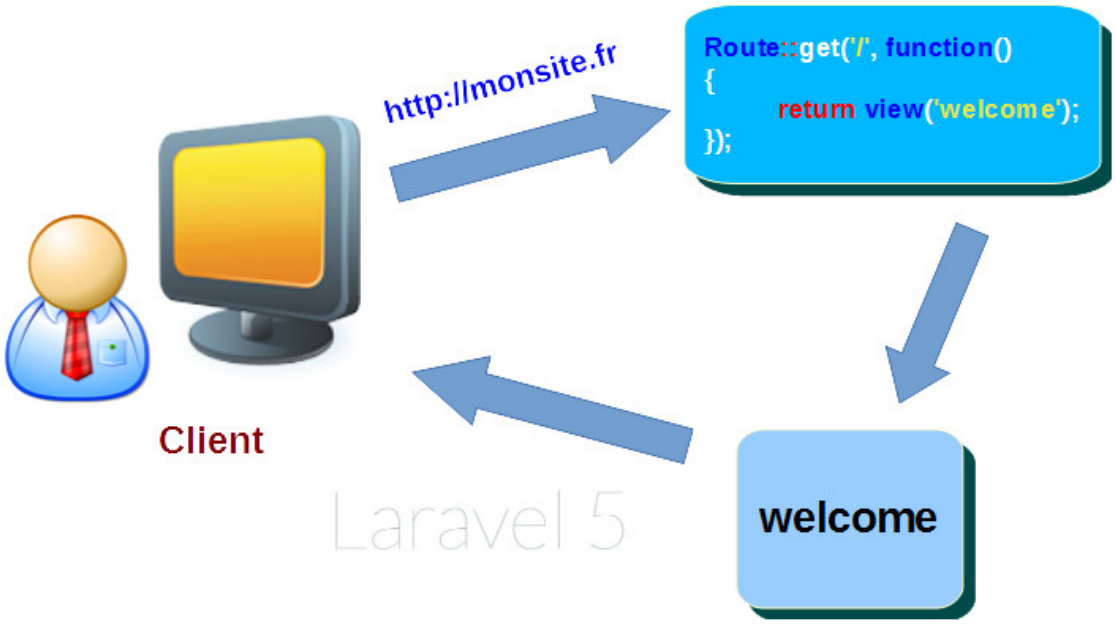
Ce fichier « welcome » se trouve bien rangé dans le dossier des vues :



C'est ce fichier comportant du code HTML qui génère le texte d'accueil que vous obtenez au démarrage initial de Laravel.

Laravel propose plusieurs *helpers* qui simplifient la syntaxe. Il y a par exemple **view** pour la classe **View** comme on l'a vu dans le code ci-dessus.

Visualisons le cycle de la requête :



Sur votre serveur local vous n'avez pas de nom de domaine et vous allez utiliser une URL de la forme ***http://localhost/tuto/public*** *en admettant que vous ayez créé Laravel dans un dossier* ***www/tuto****. Mais vous pouvez aussi créer un hôte virtuel pour avoir une situation plus réaliste comme déjà évoqué au précédent chapitre.*

Laravel accepte les verbes suivants : **get**, **post**, **put**, **patch**, **delete**, **options**, **any** (on accepte tous les verbes).

IV-B - Plusieurs routes et paramètres de route

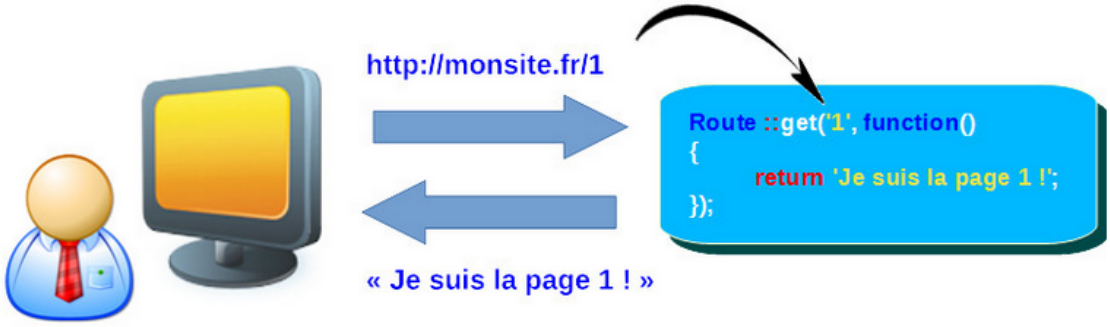
À l'installation Laravel a une seule route qui correspond à l'URL de base composée uniquement du nom de domaine. Voyons maintenant comment créer d'autres routes. Imaginons que nous ayons trois pages qui doivent être affichées avec ces URL :

1 http://monsite.fr/**1**
2 http://monsite.fr/**2**
3 http://monsite.fr/**3**

J'ai fait apparaître en gras la partie spécifique de l'URL pour chaque page. Il est facile de réaliser cela avec ce code :

```
Route::get('1', function() { return 'Je suis la page 1 !'; });
Route::get('2', function() { return 'Je suis la page 2 !'; });
Route::get('3', function() { return 'Je suis la page 3 !'; });
```

Cette fois je n'ai pas créé de vue parce que ce qui nous intéresse est uniquement une mise en évidence du routage, je retourne donc directement la réponse au client. Visualisons cela pour la page 1 :



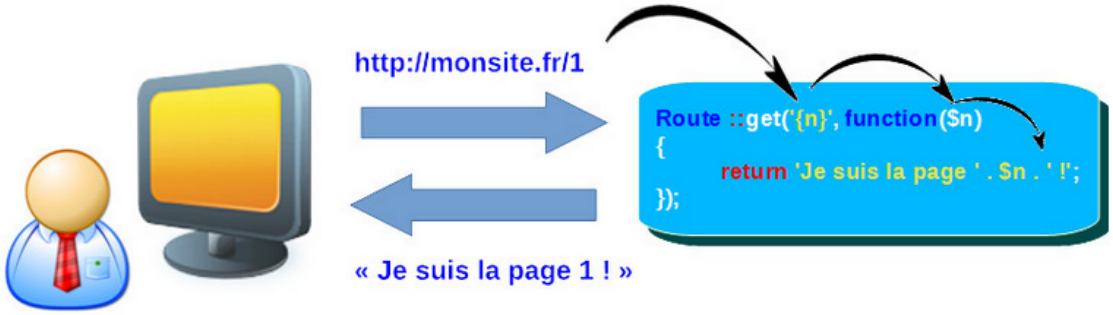
On a besoin du caractère « / » uniquement dans la route de base.

On peut maintenant se poser une question : est-il vraiment indispensable de créer trois routes alors que la seule différence tient à peu de chose : une valeur qui change ?

On peut utiliser un paramètre pour une route qui accepte des éléments variables en utilisant des accolades. Regardez ce code :

```
Route::get('{n}', function($n) {
    return 'Je suis la page ' . $n . ' !';
});
```

Et une visualisation du fonctionnement :



On dit que la route est paramétrée parce qu'elle possède un paramètre qui peut prendre n'importe quelle valeur.

On peut rendre un paramètre optionnel en lui ajoutant un point d'interrogation mais il ne doit pas être suivi par un paramètre obligatoire. Dans ce cas pour éviter une erreur d'exécution il faut prévoir une valeur par défaut pour le paramètre, par exemple :

```
Route::get('{n?}', function($n = 1) {
```

Le paramètre **n** est devenu optionnel et par défaut sa valeur est 1.

IV-C - Erreur d'exécution et contrainte de route

Dans mon double exemple précédent lorsque je dis que le résultat est le même je mens un peu. Que se passe-t-il dans les deux cas pour cette URL ?

```
http://monsite.fr/4
```

Dans le cas des trois routes vous tombez sur une erreur :

Sorry, the page you are looking for could not be found.

Par contre dans la version avec le paramètre vous obtenez une réponse valide ! Ce qui est logique parce qu'une route est trouvée. Le paramètre accepte n'importe quelle valeur et pas seulement des nombres. Par exemple avec cette URL :

```
http://monsite.fr/nimportequoi
```

Vous obtenez :

```
Je suis la page nimportequoi !
```

Ce qui vous l'avouerez n'est pas très heureux !

Pour éviter ce genre de désagrément il faut contraindre le paramètre à n'accepter que certaines valeurs. On réalise cela à l'aide d'une expression régulière :

```
Route::get('{n}', function($n) {
    return 'Je suis la page ' . $n . ' !';
})->where('n', '[1-3]');
```

Maintenant je peux affirmer que les comportements sont identiques ! Mais il nous faudra régler le problème des routes non prévues.

IV-D - Route nommée

Il est parfois utile de nommer une route, par exemple pour générer une URL ou pour effectuer une redirection. La syntaxe pour nommer une route est celle-ci :

```
Route::get('/', function() {
  return 'Je suis la page d\'accueil !';
})->name('home');
```

Par exemple pour générer l'URL qui correspond à cette route on peut utiliser l'helper **route** :

```
route('home')
```

Ce qui va générer l'URL de base du site dans ce cas : **http://monsite**.

Un avantage à utiliser des routes nommées est qu'on peut réorganiser les URL d'un site sans avoir à modifier beaucoup de code.

Nous verrons des cas d'utilisation de routes nommées dans les prochains chapitres.

IV-E - Ordre des routes

Une chose importante à connaître est l'ordre des routes !

Lisez bien ceci pour vous éviter des heures de recherches et de prises de tête. La règle est :

Les routes sont analysées dans leur ordre dans le fichier des routes.

Regardez ces deux routes :

```
1. Route::get('{n}', function($n) {
2.     return 'Je suis la page ' . $n . ' !';
3. });
4.
5. Route::get('contact', function() {
6.     return "C'est moi le contact.";
7. });
```

Que pensez-vous qu'il va se passer avec **http://monsite/contact** ?

Je vous laisse deviner et tester et surtout bien retenir ce fonctionnement !

On peut aussi grouper des routes pour simplifier la syntaxe mais nous verrons ça plus tard…

IV-F - En résumé

- Laravel possède un fichier **.htaccess** pour simplifier l'écriture des URL.
- Le système de routage est simple et explicite.
- On peut prévoir des paramètres dans les routes.
- On peut contraindre un paramètre à correspondre à une expression régulière.
- On peut nommer une route pour faciliter la génération des URL et les redirections.

V - Les réponses

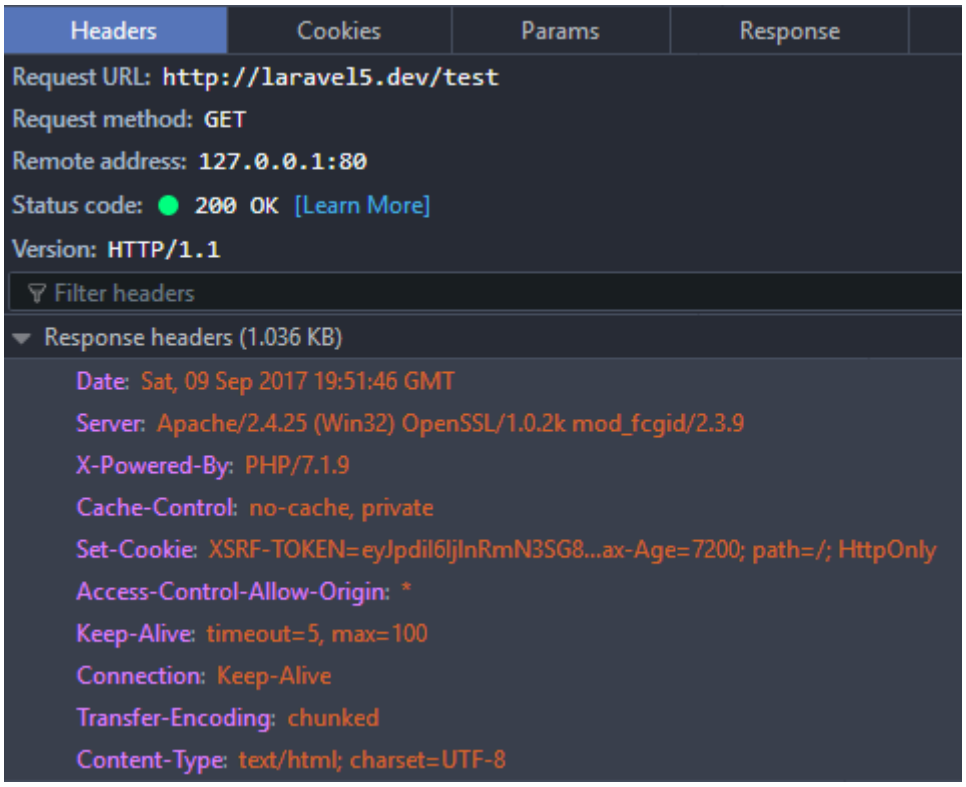
Nous avons vu précédemment comment la requête qui arrive est traitée par les routes. Voyons maintenant les réponses que nous pouvons renvoyer au client. Nous allons voir le système des vues de Laravel avec la possibilité de transmettre des paramètres. Nous verrons aussi comment créer des *templates* avec l'outil Blade.

V-A - Les réponses automatiques

Nous avons déjà construit des réponses lorsque nous avons vu le routage au chapitre précédent mais nous n'avons rien fait de spécial pour cela, juste renvoyé une chaîne de caractères comme réponse. Par exemple si nous utilisons cette route :

```
Route::get('test', function () {
    return 'un test';
});
```

Nous interceptons l'URL **http://monsite/test** et nous renvoyons la chaîne de caractères « un test ». Mais évidemment Laravel en coulisse construit une véritable réponse HTTP. Voyons cela :



On se rend compte qu'on a une requête complète avec ses headers mais nous ne pouvons pas intervenir sur ces valeurs. Remarquez au passage qu'on a des cookies, on en reparlera lorsque nous verrons les sessions.

*Le **content-type** indique le type **MIME** du document retourné, pour que le navigateur sache quoi faire du document en fonction de la nature de son contenu. Par exemple :*

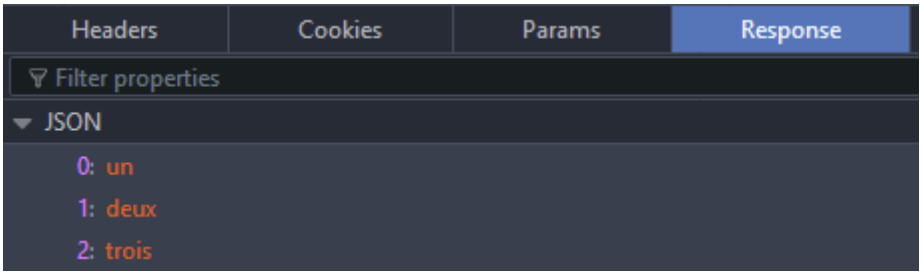
- ***text/html** : page HTML classique ;*
- ***text/plain** : simple texte sans mise en forme ;*
- ***application/pdf** : fichier PDF ;*
- ***application/json** : données au format JSON ;*
- ***application/octet-stream** : téléchargement de fichier.*

Pour une liste exhaustive c'est ici.

Maintenant voyons ce que ça donne si on renvoie un tableau :

```
Route::get('test', function () {
    return ['un', 'deux', 'trois'];
});
```

Cette fois on reçoit une réponse JSON :



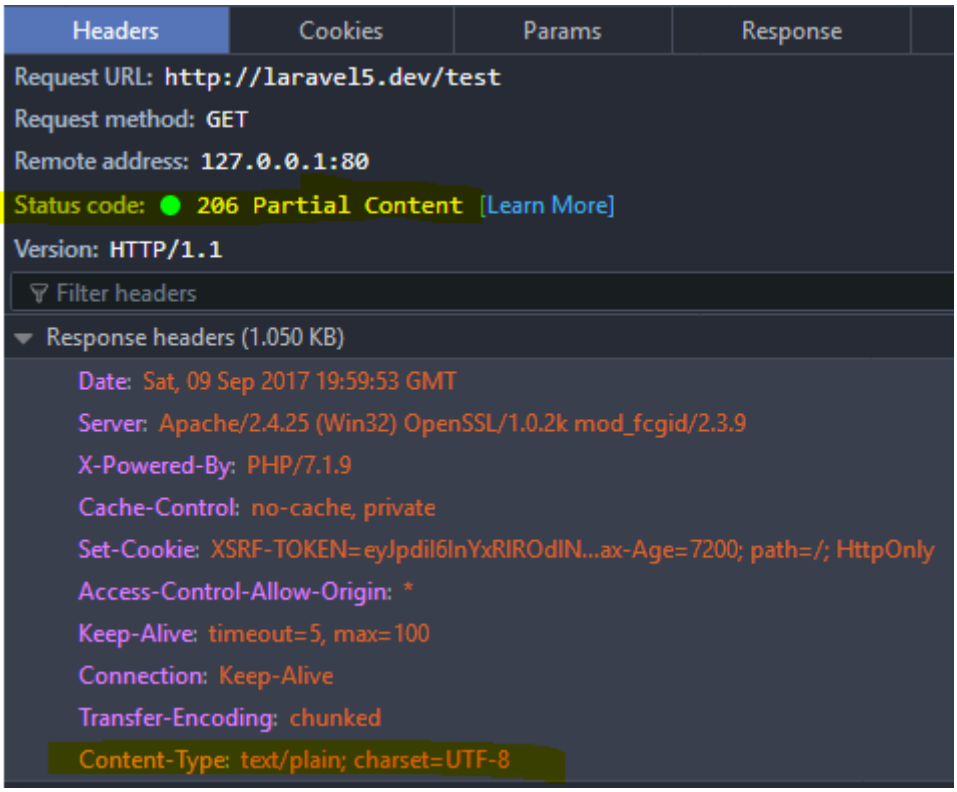
Donc si vous voulez renvoyer du JSON il suffit de retourner un tableau et Laravel s'occupe de tout !

V-B - Construire une réponse

Le fonctionnement automatique c'est bien mais parfois on veut imposer des valeurs. Dans ce cas il faut utiliser une classe de Laravel pour construire une réponse. Comme la plupart du temps on a un helper qui nous évite de déclarer la classe en question (en l'occurrence c'est la classe **Illuminate\Http\Response** qui hérite de celle de Symfony : **Symfony\Component\HttpFoundation\Response**).

```
Route::get('test', function () {
    return response('un test', 206)->header('Content-Type', 'text/plain');
});
```

Cette fois j'impose un code (**206** : envoi partiel) et un type MIME (**text/plain**) :



Dans le protocole HTTP il existe des codes pour spécifier les réponses. Ces codes sont classés par grandes catégories. Voici les principaux :

- ***200** : requête exécutée avec succès ;*
- ***403** : ressource interdite ;*
- ***404** : la ressource demandée n'a pas été trouvée ;*
- ***503** : serveur indisponible.*

Pour une liste complète c'est ici.

En fait vous aurez rarement la nécessité de préciser les headers parce que Laravel s'en charge très bien, mais vous voyez que c'est facile à faire.

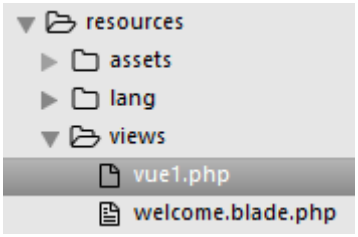
On peut aussi ajouter un cookie avec la méthode **cookie**.

V-C - Les vues

Dans une application réelle vous retournerez rarement la réponse directement à partir d'une route, vous passerez au moins par une vue. Dans sa version la plus simple une vue est un simple fichier avec du code HTML :

```
1. <!doctype html>
2. <html lang="fr">
3. <head>
4.     <meta charset="UTF-8">
5.     <title>Ma première vue</title>
6. </head>
7. <body>
8.     Je suis une vue !
9. </body>
10. </html>
```

Il faut enregistrer cette vue (j'ai choisi le nom « vue1 ») dans le dossier **resources/views** avec l'extension php :



Même si vous ne mettez que du code HTML dans une vue vous devez l'enregistrer avec l'extension php.

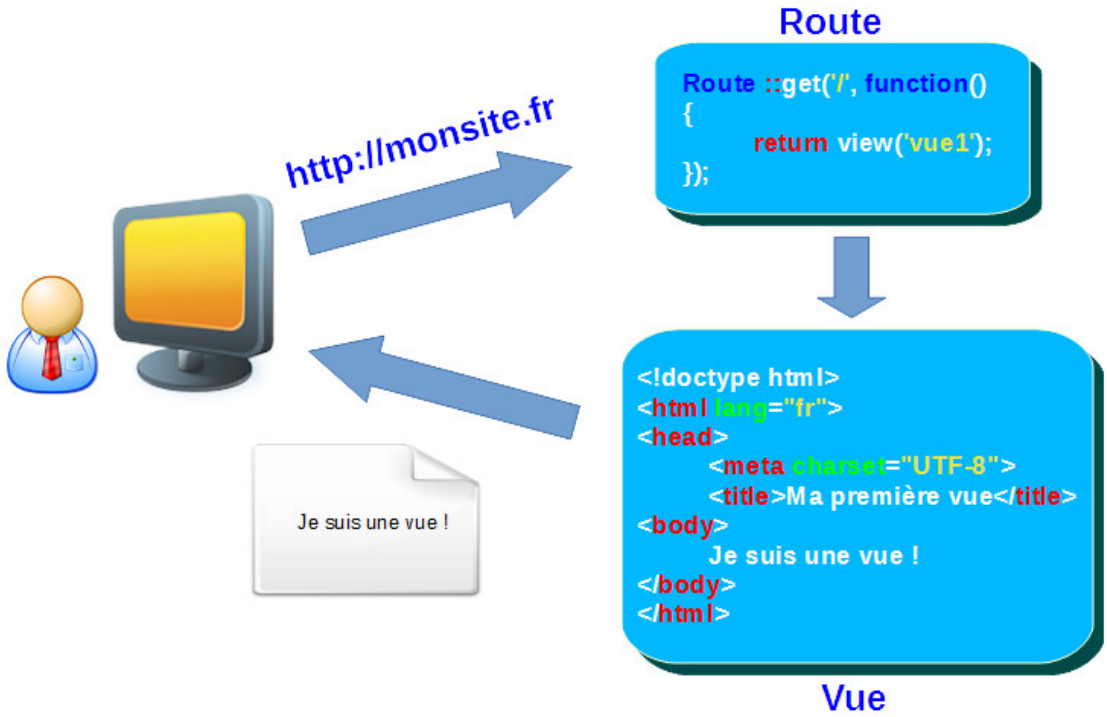
On peut appeler cette vue à partir d'une route avec ce code :

```
Route::get('/', function() {
    return view('vue1');
});
```

Pour que ça fonctionne commentez ou supprimez la route de base de l'installation.

Je vous rappelle la belle sémantique de Laravel qui se lit comme de la prose : je retourne (**return**) une vue (**view**) à partir du fichier de vue « vue1 ».

Voici une illustration du processus :



V-C-1 - Vue paramétrée

En général on a des informations à transmettre à une vue, voyons à présent comment mettre cela en place. Supposons que nous voulions répondre à ce type de requête :

```
http://monsite.fr/article/n
```

Le paramètre **n** pouvant prendre une valeur numérique, voyons comment cette URL est constituée :

- la base de l'URL est constante pour le site, quelle que soit la requête ;
- la partie fixe ici correspond aux articles ;
- la partie variable correspond au numéro de l'article désiré (le paramètre).

V-C-2 - Route

Il nous faut une route pour intercepter ces URL :

```
Route::get('article/{n}', function($n) {
    return view('article')->with('numero', $n);
})->where('n', '[0-9]+');
```

On transmet la variable à la vue avec la méthode **with**.

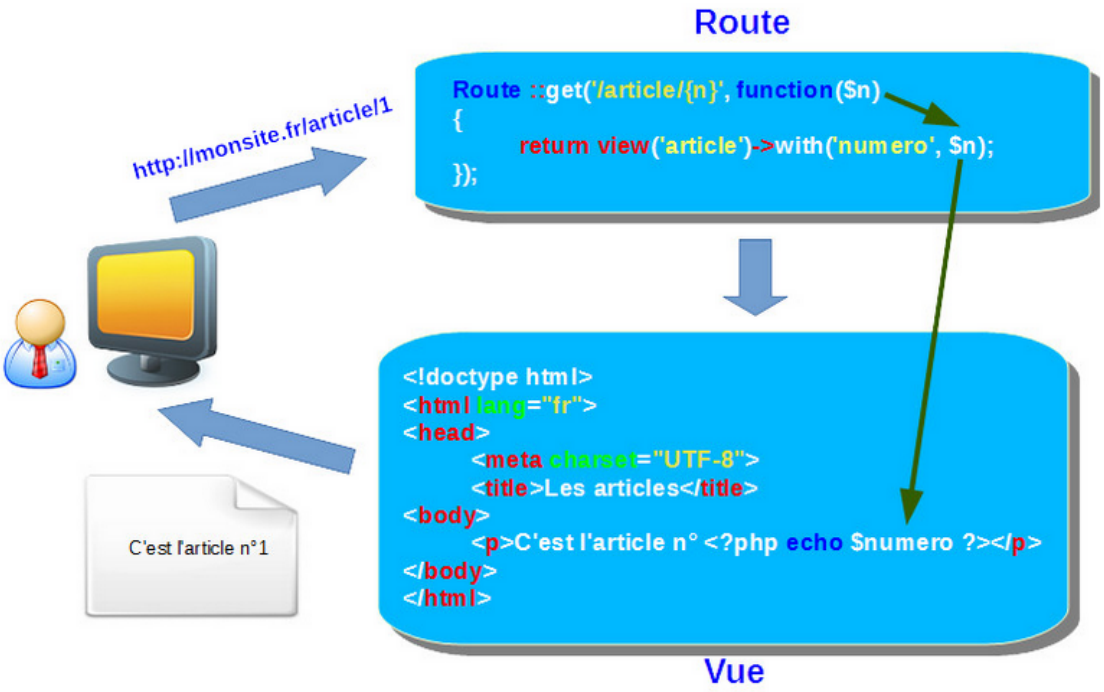
V-C-3 - Vue

Il ne nous reste plus qu'à créer la vue **article.php** dans le dossier **resources/views** :

```
1. <!doctype html>
2. <html lang="fr">
3. <head>
4.     <meta charset="UTF-8">
5.     <title>Les articles</title>
6. </head>
7. <body>
8.     <p>C'est l'article n° <?php echo $numero ?></p>
9. </body>
10. </html>
```

Pour récupérer le numéro de l'article on utilise la variable **$numero**.

Voici une schématisation du fonctionnement :



*Il existe une méthode « magique » pour transmettre un paramètre, par exemple pour transmettre la variable **numero** comme je l'ai fait ci-dessus on peut écrire le code ainsi :*

```
return view('article')->withNumero($n);
```

Il suffit de concaténer le nom de la variable au mot clé **with**.

On peut aussi transmettre un tableau comme paramètre :

```
return view('article', ['numero' => $n]);
```

V-D - Blade

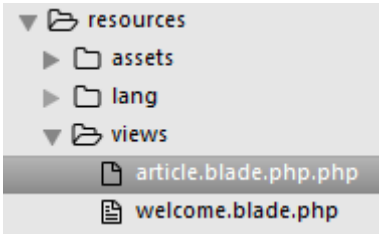
Laravel possède un moteur de template élégant nommé Blade qui nous permet de faire pas mal de choses. La première est de nous simplifier la syntaxe. Par exemple au lieu de la ligne suivante que nous avions dans la vue précédente :

```
<p>C'est l'article n° <?php echo $numero ?></p>
```

On peut utiliser cette syntaxe avec Blade :

```
<p>C'est l'article n° {{ $numero }}</p>
```

Tout ce qui se trouve entre les doubles accolades est interprété comme du code PHP. Mais pour que ça fonctionne il faut indiquer à Laravel qu'on veut utiliser Blade pour cette vue. Ça se fait simplement en modifiant le nom du fichier :



Il suffit d'ajouter « blade » avant l'extension « php ». Vous pouvez tester l'exemple précédent avec ces modifications et vous verrez que tout fonctionne parfaitement avec une syntaxe épurée.

Il y a aussi la version avec la syntaxe {!! … !!}. La différence entre les deux versions est que le texte entre les doubles accolades est échappé ou purifié. C'est une mesure de sécurité parce qu'un utilisateur pourrait très bien mettre du code malicieux dans l'URL.

V-E - Un template

Une fonction fondamentale de Blade est de permettre de faire du templating, c'est-à-dire de factoriser du code de présentation. Poursuivons notre exemple en complétant notre application avec une autre route chargée d'intercepter des URL pour des factures. Voici la route :

```
Route::get('facture/{n}', function($n) {
    return view('facture')->withNumero($n);
})->where('n', '[0-9]+');
```

Et voici la vue :

```
1. <!doctype html>
2. <html lang="fr">
3. <head>
4.     <meta charset="UTF-8">
5.     <title>Les factures</title>
6. </head>
7. <body>
8.     <p>C'est la facture n° {{ $numero }}</p>
9. </body>
10. </html>
```

On se rend compte que cette vue est pratiquement la même que celle des articles. Il serait intéressant de placer le code commun dans un fichier.

C'est justement le but d'un template d'effectuer cette opération.

Voici le template :

```
1. <!doctype html>
2. <html lang="fr">
3. <head>
4.     <meta charset="UTF-8">
5.     <title>@yield('titre')</title>
6. </head>
7. <body>
8.     @yield('contenu')
9. </body>
10. </html>
```

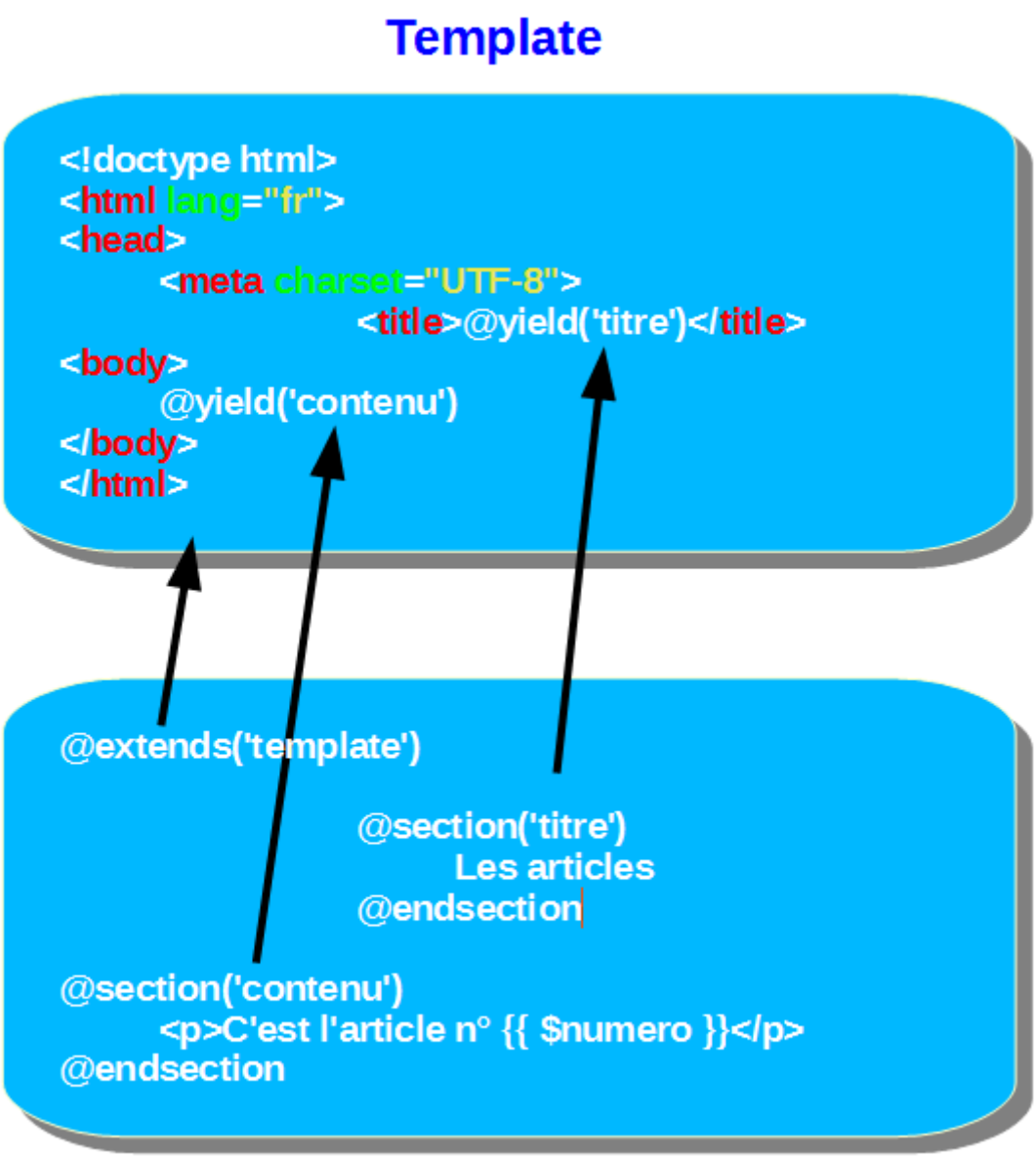
J'ai repris le code commun et prévu deux emplacements repérés par le mot clé **@yield** et nommés « titre » et « contenu ». Il suffit maintenant de modifier les deux vues. Voilà pour les articles :

```
@extends('template')

@section('titre')
    Les articles
@endsection

@section('contenu')
    <p>C'est l'article n° {{ $numero }}</p>
@endsection
```

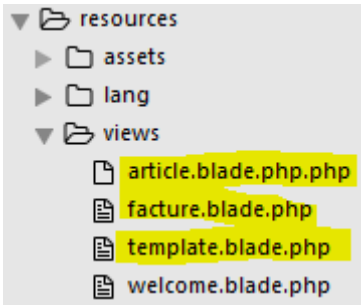
Et voilà pour les factures :

```
@extends('template')

@section('titre')
    Les factures
@endsection

@section('contenu')
    <p>C'est la facture n° {{ $numero }}</p>
@endsection
```

Dans un premier temps on dit qu'on veut utiliser le template avec **@extends** et le nom du template « template ». Ensuite on remplit les zones prévues dans le template grâce à la syntaxe **@section** en précisant le nom de l'emplacement et en fermant avec **@endsection**. Voici un schéma pour bien visualiser tout ça avec les articles :



Au niveau du dossier des vues on a donc les trois fichiers :



Blade permet de faire bien d'autres choses, nous verrons cela dans les prochains chapitres.

Lorsqu'elles deviendront nombreuses on organisera nos vues dans des dossiers.

V-F - Les redirections

Souvent il ne faut pas envoyer directement la réponse mais rediriger sur une autre URL. Pour réaliser cela on a l'helper **redirect** :

```
return redirect('facture');
```

Ici on redirige sur l'URL **http://monsite/facture**.

On peut aussi rediriger sur une route nommée. Par exemple vous avez cette route :

```
Route::get('users/action', function() {
    return view('users.action');
})->name('action');
```

Cette route s'appelle **action** et elle correspond à l'URL **http://monsite/users/action**. On peut simplement rediriger sur cette route avec cette syntaxe :

```
return redirect()->route('action');
```

Si la route comporte un paramètre (ou plusieurs) on peut aussi lui assigner une valeur. Par exemple avec cette route :

```
Route::get('users/action/{type}', function($type) {
    return view('users.action');
})->name('action');
```

On peut rediriger ainsi en renseignant le paramètre :

```
return redirect()->route('action', ['type' => 'play']);
```

Parfois on veut tout simplement recharger la même page, par exemple lors de la soumission d'un formulaire avec des erreurs dans la validation des données, il suffit alors de faire :

```
return back();
```

On verra de nombreux exemples de redirections dans les prochains chapitres.

V-G - En résumé

- Laravel construit automatiquement des réponses HTTP lorsqu'on retourne une chaîne de caractère ou un tableau.
- Laravel offre la possibilité de créer des vues.
- Il est possible de transmettre simplement des paramètres aux vues.
- L'outil Blade permet de créer des templates et d'optimiser ainsi le code des vues.
- On peut facilement effectuer des redirections avec transmission éventuelle de paramètres.

VI - Artisan et les contrôleurs

Nous avons vu le cycle d'une requête depuis son arrivée, son traitement par les routes et sa réponse avec des vues qui peuvent être boostées par Blade. Avec tous ces éléments vous pourriez très bien réaliser un site web complet mais Laravel offre encore bien des outils performants que je vais vous présenter.

Pour correctement organiser son code dans une application Laravel il faut bien répartir les tâches. Dans les exemples vus jusqu'à présent j'ai renvoyé une vue à partir d'une route, vous ne ferez jamais cela dans une application réelle

(même si personne ne vous empêchera de le faire !). Les routes sont juste un système d'aiguillage pour trier les requêtes qui arrivent.

Mais alors qui s'occupe de la suite ?

Et bien ce sont les contrôleurs, le sujet de ce chapitre.

Nous allons aussi découvrir l'outil **Artisan** qui est la boîte à outils du développeur pour Laravel.

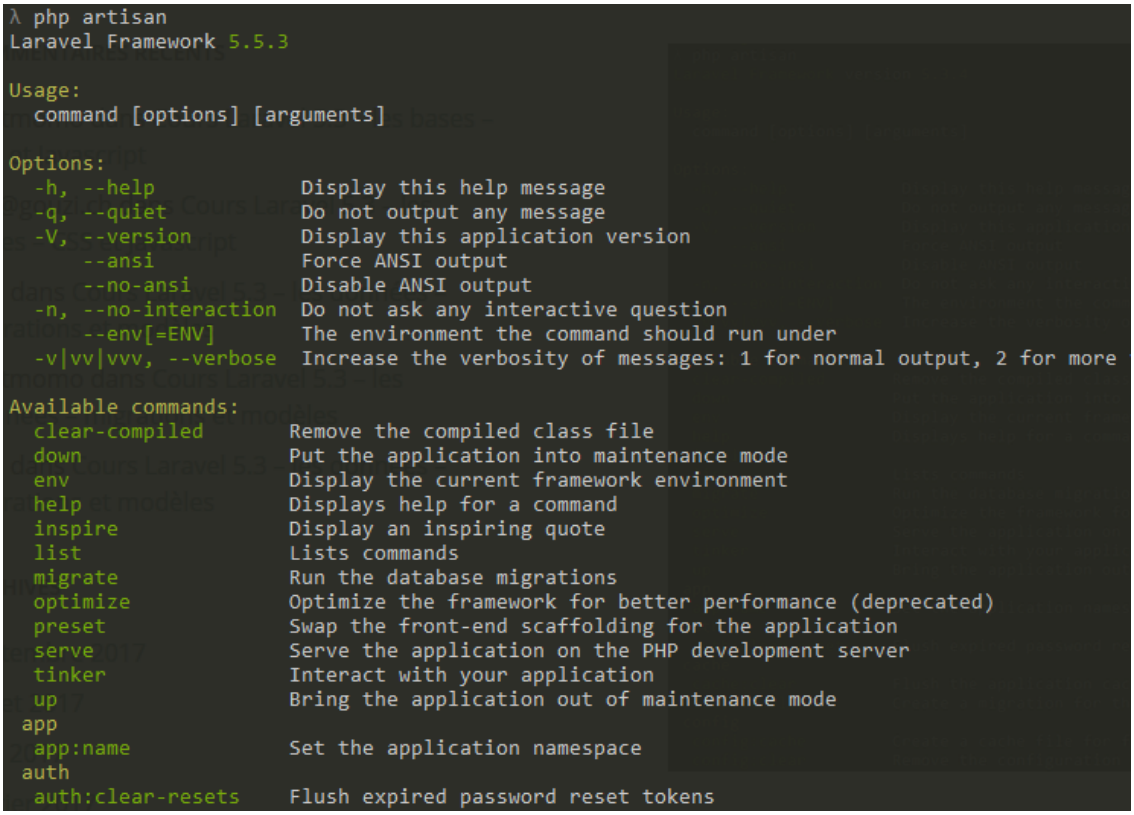
VI-A - Artisan

Lorsqu'on construit une application avec Laravel on a de nombreuses tâches à accomplir, comme par exemple créer des classes ou vérifier les routes.

C'est là qu'intervient Artisan, le compagnon indispensable. Il fonctionne en ligne de commande, donc à partir de la console. Il suffit de se positionner dans le dossier racine et d'utiliser la commande :

```
php artisan
```

pour obtenir la liste de ses possibilités, en voici un extrait :



Nous verrons peu à peu les principales commandes disponibles. Il y en a une pour connaître les routes prévues dans le code. Voici ce que ça donne avec une nouvelle installation :

```
λ php artisan route:list
+--------+----------+----------+------+---------+--------------+
| Domain | Method   | URI      | Name | Action  | Middleware   |
+--------+----------+----------+------+---------+--------------+
|        | GET|HEAD | /        |      | Closure | web          |
|        | GET|HEAD | api/user |      | Closure | api,auth:api |
+--------+----------+----------+------+---------+--------------+
```

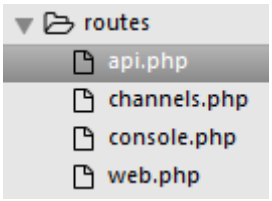
On sait que la seule route au départ est celle-ci :

```
Route::get('/', function () {
     return view('welcome');
 });
```

Elle correspond à la première ligne de la liste.

Mais à quoi correspond la seconde route ?

On a vu qu'il y a quatre fichiers de routes, en particulier on a celui pour les API :



Lui aussi comporte une route par défaut :

```
Route::middleware('auth:api')->get('/user', function (Request $request) {
    return $request->user();
});
```

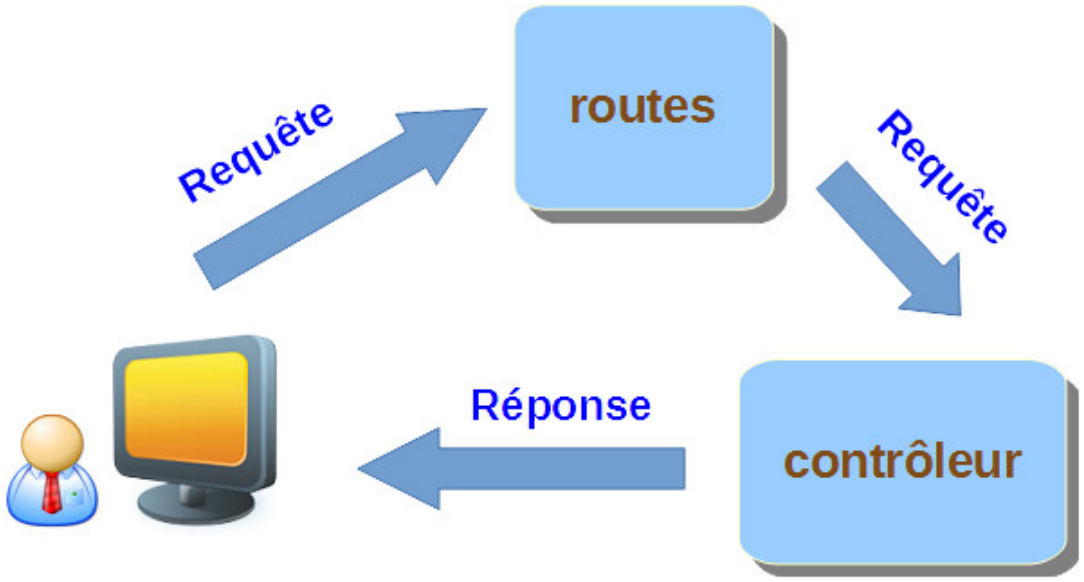
On ne va pas s'y intéresser dans ce cours. Vous pouvez commenter ou supprimer cette route pour éviter de polluer la liste.

Je parlerai des *middlewares* dans un prochain chapitre.

VI-B - Les contrôleurs

VI-B-1 - Rôle

La tâche d'un contrôleur est de réceptionner une requête (qui a déjà été sélectionnée par une route) et de définir la réponse appropriée, rien de moins et rien de plus. Voici une illustration du processus :

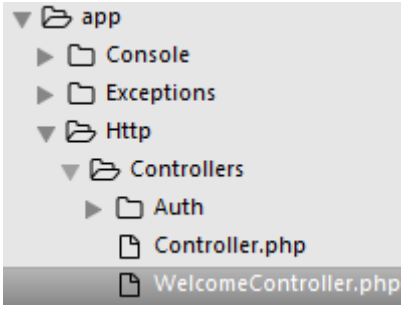


VI-B-2 - Constitution

Pour créer un contrôleur nous allons utiliser Artisan. Dans la console entrez cette commande :

```
php artisan make:controller WelcomeController
```

Si tout se passe bien vous allez trouver le contrôleur ici :



Avec ce code :

```
1. <?php
2.
3. namespace App\Http\Controllers;
4.
5. use Illuminate\Http\Request;
6.
7. class WelcomeController extends Controller
8. {
9.     //
10. }
```

Ajoutez la méthode **index** :

```
1. <?php
2. ...
3. class WelcomeController extends Controller
4. {
5.     public function index()
6.     {
```

```
7.          return view('welcome');
8.      }
9.  }
```

Analysons un peu le code :

- on trouve en premier l'espace de nom (**App\Http\Controllers**) ;
- le contrôleur hérite de la classe **Controller** qui se trouve dans le même dossier et qui permet de factoriser des actions communes à tous les contrôleurs ;
- on trouve enfin la méthode **index** qui renvoie quelque chose que maintenant vous connaissez : une vue, en l'occurrence « welcome » dont nous avons déjà parlé. Donc si j'appelle cette méthode je retourne la vue « welcome » au client.

VI-B-3 - Liaison avec les routes

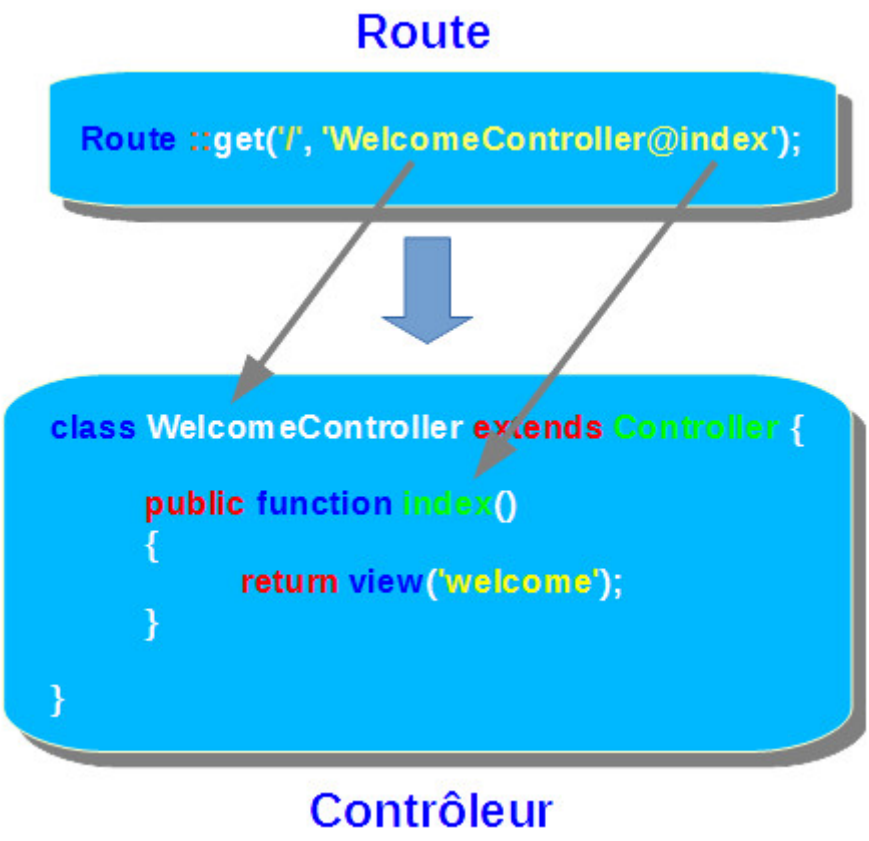
Maintenant la question qu'on peut se poser est : comment s'effectue la liaison entre les routes et les contrôleurs ?

Ouvrez le fichier des routes et entrez ce code :

```
Route::get('/', 'WelcomeController@index');
```

Maintenant avec l'URL de base vous devez retrouver la page d'accueil de Laravel :

Voici une visualisation de la liaison entre la route et le contrôleur :



On voit qu'au niveau de la route il suffit de désigner le nom du contrôleur et le nom de la méthode séparés par @.

Si vous êtes attentif au code vous avez sans doute remarqué qu'au niveau de la route on ne spécifie pas l'espace de noms du contrôleur, on peut légitimement se demander comment on le retrouve. Laravel nous simplifie la syntaxe en ajoutant automatiquement cet espace de nom.

VI-B-4 - Route nommée

De la même manière que nous pouvons nommer une route classique on peut aussi donner un nom à une route qui pointe une méthode de contrôleur :

```
Route::get('/', 'WelcomeController@index')->name('home');
```

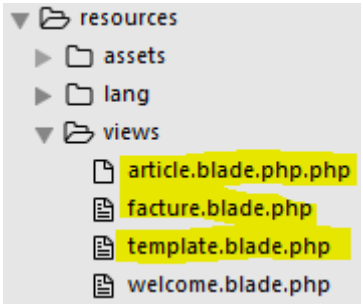
Si on utilise Artisan pour lister les routes :

```
λ php artisan route:list
+--------+----------+-----+------+-----------------------------------------+------------+
| Domain | Method   | URI | Name | Action                                  | Middleware |
+--------+----------+-----+------+-----------------------------------------+------------+
|        | GET|HEAD | /   | home | App\Http\Controllers\WelcomeController@index | web        |
+--------+----------+-----+------+-----------------------------------------+------------+
```

On voit bien que l'action est faite par le contrôleur avec précision de la méthode à utiliser. On trouve aussi le nom de la route.

VI-C - Utilisation d'un contrôleur

Voyons maintenant un exemple pratique de mise en œuvre d'un contrôleur. On va conserver notre exemple avec les articles mais maintenant traité avec un contrôleur. On conserve le même template et les mêmes vues :



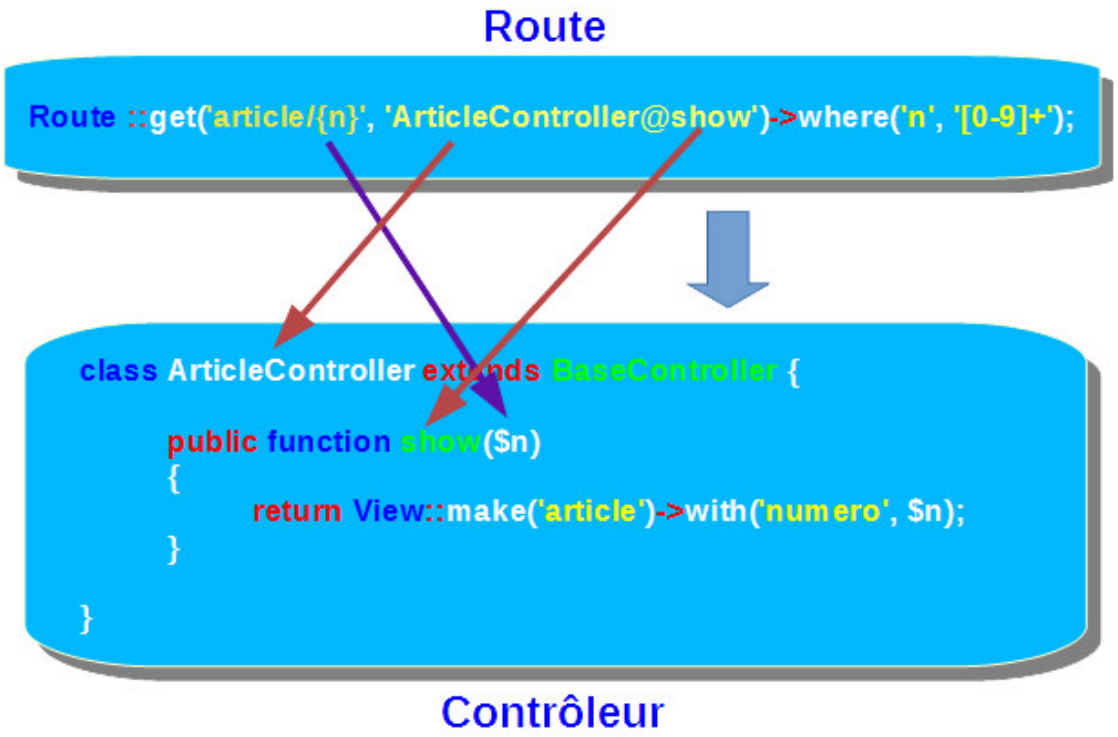
On va créer un contrôleur (entraînez-vous à utiliser Artisan) pour les articles :

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;

class ArticleController extends Controller
{
    public function show($n)
    {
        return view('article')->with('numero', $n);
    }
}
```

Dans ce contrôleur on a une méthode **show** chargée de générer la vue. Il ne nous reste plus qu'à créer la route :

```php
Route::get('article/{n}', 'ArticleController@show')->where('n', '[0-9]+');
```

Voici une illustration du fonctionnement avec ce contrôleur :



Notez qu'on pourrait utiliser la méthode « magique » pour la transmission du paramètre à la vue :

```
return view('article')->withNumero($n);
```

VI-D - En résumé

- Les contrôleurs servent à réceptionner les requêtes triées par les routes et à fournir une réponse au client.
- Artisan permet de créer facilement un contrôleur.
- Il est facile d'appeler une méthode de contrôleur à partir d'une route.
- On peut nommer une route qui pointe vers une méthode de contrôleur.

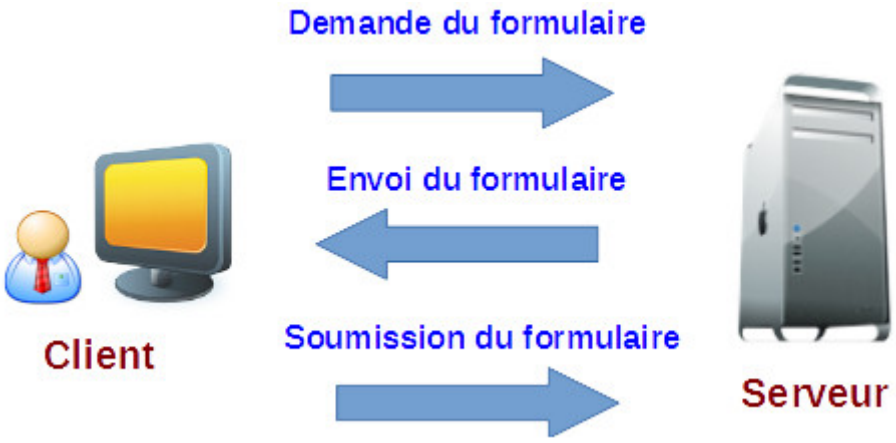
VII - Formulaires et middlewares

Dans bien des circonstances, le client envoie des informations au serveur. La situation la plus générale est celle d'un formulaire. Nous allons voir dans ce chapitre comment créer facilement un formulaire avec Laravel, comment réceptionner les entrées et nous améliorerons notre compréhension du routage.

Nous verrons aussi l'importante notion de middleware.

VII-A - Scénario et routes

Nous allons envisager un petit scénario avec une demande de formulaire de la part du client, sa soumission et son traitement :



On va donc avoir besoin de deux routes :

- une pour la demande du formulaire avec une méthode **get ;**
- une pour la soumission du formulaire avec une méthode **post**.

On va donc créer ces deux routes dans le fichier **routes/web.php** :

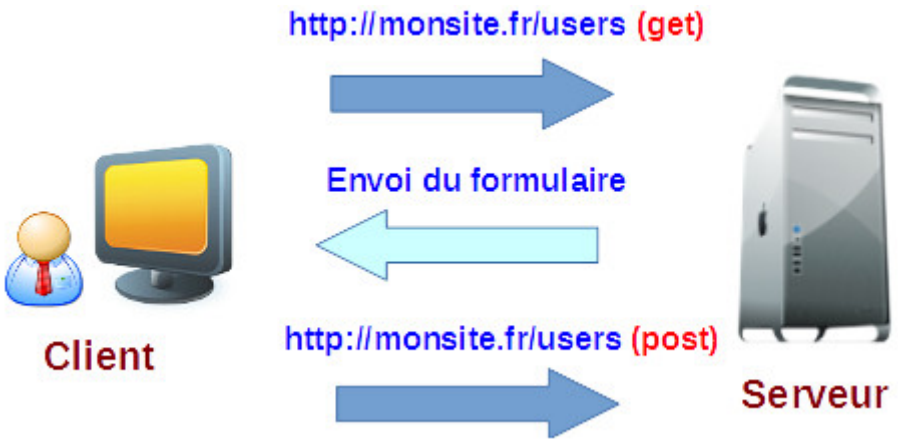
```
Route::get('users', 'UsersController@create');
Route::post('users', 'UsersController@store');
```

Jusque-là on avait vu seulement des routes avec le verbe **get**, on a maintenant aussi une route avec le verbe **post**.

Les URL correspondantes sont donc :

- **http://monsite.fr/users** avec la méthode **get** ;
- **http://monsite.fr/users** avec la méthode **post**.

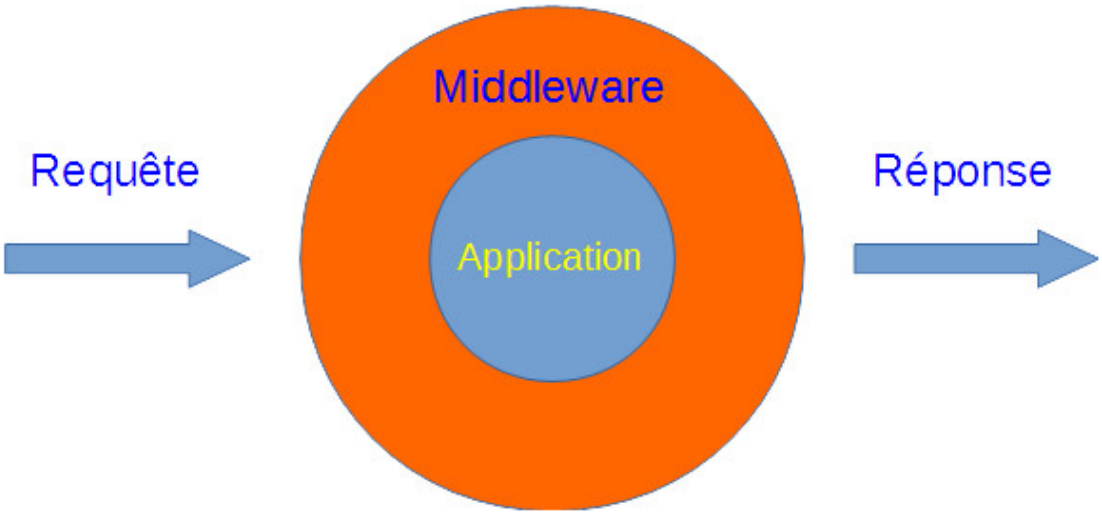
Donc on a la même URL, seul le verbe diffère. Voici le scénario schématisé avec les URL :



VII-B - Les middlewares

Les middlewares sont chargés de filtrer les requêtes HTTP qui arrivent dans l'application, ainsi que celles qui en partent (beaucoup moins utilisées). Le cas le plus classique est celui qui concerne la vérification de l'authentification d'un utilisateur pour qu'il puisse accéder à certaines ressources. On peut aussi utiliser un middleware par exemple pour démarrer la gestion des sessions.

Voici un schéma pour illustrer cela :



On peut avoir en fait plusieurs middlewares en pelures d'oignon, chacun effectue son traitement et transmet la requête ou la réponse au suivant.

Donc dès qu'il y a un traitement à faire à l'arrivée des requêtes (ou à leur départ) un middleware est tout indiqué.

Laravel peut servir comme application « web » ou comme « api ». Dans le premier cas on a besoin :

- de gérer les cookies ;
- de gérer une session ;
- de gérer la protection CSRF (dont je parle plus loin dans ce chapitre).

Si vous regardez dans le fichier **app/Http/Kernel.php** :

```
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        // \Illuminate\Session\Middleware\AuthenticateSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],

    'api' => [
        'throttle:60,1',
        'bindings',
    ],
];
```

On trouve les deux middlewares de groupes (ils rassemblent plusieurs middlewares) « web » et « api ». On voit que dans le premier cas on active bien les cookies, les sessions et la vérification CSRF.

Par défaut toutes les routes que vous entrez dans le fichier **routes/web.php** sont incluses dans le groupe « web ». Si vous regardez dans le provider **app/Providers/RouteServiceProvider.php** vous trouvez cette inclusion :

```
protected function mapWebRoutes()
{
    Route::middleware('web')
         ->namespace($this->namespace)
         ->group(base_path('routes/web.php'));
}
```

VII-C - Le formulaire

Pour faire les choses correctement nous allons prévoir un template **resources/views/template.blade.php** :

```
<!doctype html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
</head>
<body>
    @yield('contenu')
</body>
</html>
```

Et une vue **resources/views/infos.blade.php** qui utilise ce template :

```
@extends('template')

@section('contenu')
    <form action="{{ url('users') }}" method="POST">
        {{ csrf_field() }}
        <label for="nom">Entrez votre nom : </label>
        <input type="text" name="nom" id="nom">
        <input type="submit" value="Envoyer !">
    </form>
@endsection
```

Je vous parle plus loin de la signification de la syntaxe {{ csrf_field() }}.

*L'helper **url** est utilisé pour générer l'URL complète pour l'action du formulaire. Pour une route nommée on pourrait utiliser l'helper **route**.*

Le résultat sera un formulaire sans fioriture :

Entrez votre nom : Envoyer !

VII-D - Laravel Collective ?

Ceux qui ont suivi mes cours précédents seront sans doute surpris de ne pas trouver ici et dans les chapitres suivants l'utilisation de **la librairie HTML de Laravel Collective**.

Cette librairie faisait à l'origine partie de Laravel dans sa version 4. À partir de la version 5 elle a été retirée du projet et récupérée en tant que librairie indépendante. Comme j'étais habitué à ce composant je l'ai naturellement intégré à mes projets et conseillé dans mes cours.

Le temps passant je me suis rendu compte que je préfère personnellement coder mes vues avec la syntaxe classique plutôt que de la masquer avec les méthodes de ce composant malgré les avantages en matière de concision du code et de rapidité de développement. C'est un choix personnel.

Si vous voulez utiliser cette librairie il vous suffit de suivre **la procédure d'installation détaillée ici** et ensuite de consulter la documentation qui suit et qui est très bien faite. Par exemple pour notre formulaire on se retrouve avec cette syntaxe :

```
1. @extends('template')
2.
3. @section('contenu')
4.     {!! Form::open(['url' => 'users']) !!}
5.         {!! Form::label('nom', 'Entrez votre nom : ') !!}
6.         {!! Form::text('nom') !!}
7.         {!! Form::submit('Envoyer !') !!}
8.     {!! Form::close() !!}
9. @endsection
```

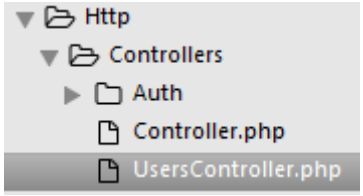
Libre à vous d'adapter les formulaires de ce cours avec cette librairie si elle vous convient mais je n'y ferai plus référence dans les chapitres suivants au profit d'une syntaxe classique.

VII-E - Le contrôleur

Il ne nous manque plus que le contrôleur pour faire fonctionner tout ça. Utilisez Artisan pour générer un contrôleur :

```
php artisan make:controller UsersController
```

Vous devez le retrouver ici :



Modifiez ensuite son code ainsi :

```
1. <?php
2.
3. namespace App\Http\Controllers;
4.
5. use Illuminate\Http\Request;
6.
7. class UsersController extends Controller
8. {
```

```
9.      public function create()
10.      {
11.          return view('infos');
12.      }
13.
14.      public function store(Request $request)
15.      {
16.          return 'Le nom est ' . $request->input('nom');
17.      }
18. }
```

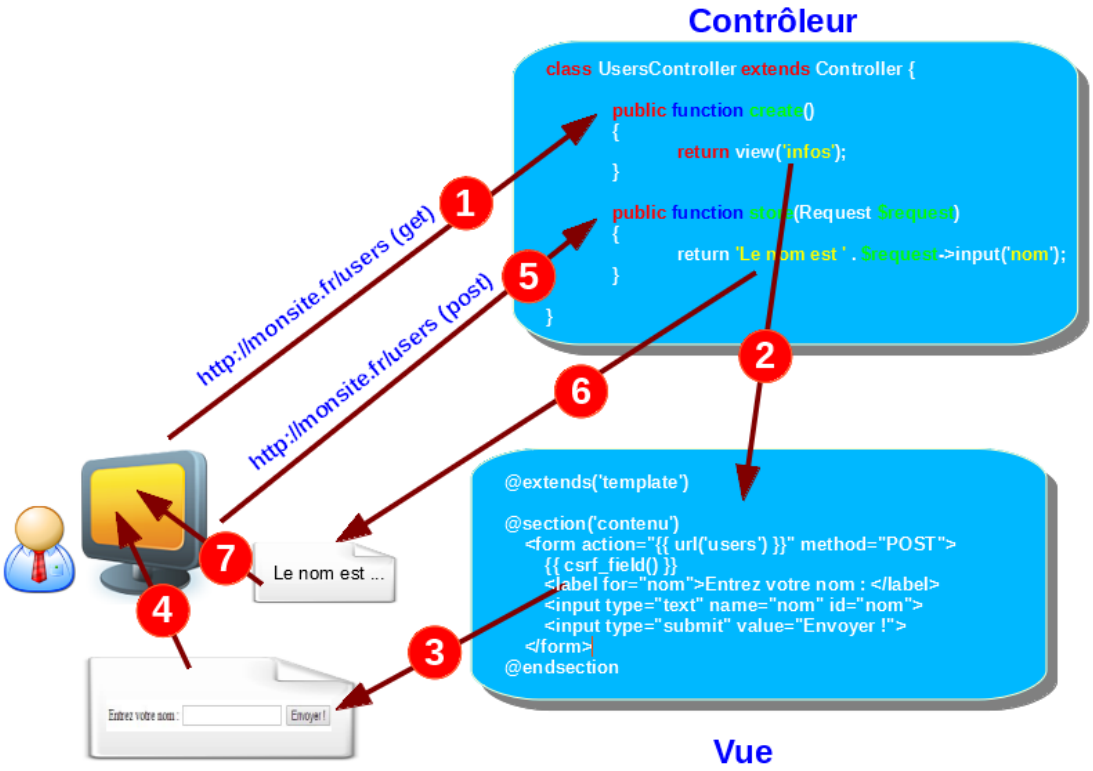
Le contrôleur possède deux méthodes :

- la méthode **create** qui reçoit l'URL **http://monsite.fr/users** avec le verbe **get** et qui retourne le formulaire ;
- la méthode **store** qui reçoit l'URL **http://monsite.fr/users** avec le verbe **post** et qui traite les entrées.

Pour la première méthode il n'y a rien de nouveau et je vous renvoie aux chapitres précédents si quelque chose ne vous paraît pas clair. Par contre nous allons nous intéresser à la seconde méthode.

Dans cette seconde méthode on veut récupérer l'entrée du client. Encore une fois la syntaxe est limpide : on veut dans la requête (**request**) les entrées (**input**) récupérer celle qui s'appelle **nom**.

Si vous faites fonctionner tout ça vous devez finalement obtenir l'affichage du nom saisi. Voici une schématisation du fonctionnement qui exclut les routes pour simplifier :



(1) le client envoie la requête de demande du formulaire qui est transmise au contrôleur par la route (non représentée sur le schéma),
(2) le contrôleur crée la vue « infos »,
(3) la vue « infos » crée le formulaire,
(4) le formulaire est envoyé au client,
(5) le client soumet le formulaire, le contrôleur reçoit la requête de soumission par l'intermédiaire de la route (non représentée sur le schéma),
(6) le contrôleur génère la réponse,

(7) la réponse est envoyée au client.

VII-F - La protection CSRF

On a vu que le formulaire généré par Laravel comporte une ligne un peu particulière :

```
{{ csrf_field() }}
```

Si on regarde le code généré on trouve quelque chose dans ce genre :

```
<input type="hidden" name="_token" value="iIjW9PMNsV6VKT2sIc16ShoTf6SdYVZolVUGsxDI">
```

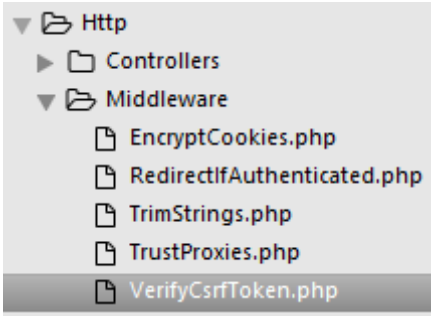
À quoi cela sert-il ?

Tout d'abord **CSRF** signifie **Cross-Site Request Forgery**. C'est une attaque qui consiste à faire envoyer par un client une requête à son insu. Cette attaque est relativement simple à mettre en place et consiste à envoyer à un client authentifié sur un site un script dissimulé (dans une page web ou un email) pour lui faire accomplir une action à son insu.

Pour se prémunir contre ce genre d'attaque Laravel génère une valeur aléatoire (**token**) associée au formulaire de telle sorte qu'à la soumission cette valeur est vérifiée pour être sûr de l'origine.

Vous vous demandez peut-être où se trouve ce middleware CSRF ?

Il est bien rangé dans le dossier **app/Http/Middleware** :



Pour tester l'efficacité de cette vérification, essayez un envoi de formulaire sans le token en modifiant ainsi la vue :

```
1. @extends('template')
2.
3. @section('contenu')
4.     <form action="{{ url('users') }}" method="POST">
5.         <label for="nom">Entrez votre nom : </label>
6.         <input type="text" name="nom" id="nom">
7.         <input type="submit" value="Envoyer !">
8.     </form>
9. @endsection
```

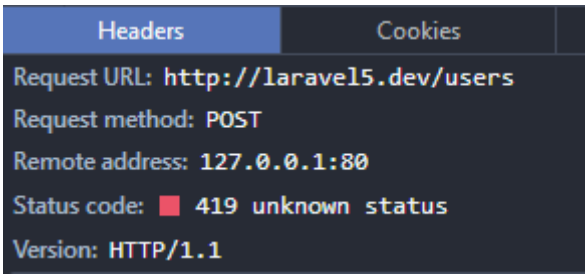
Vous tombez sur cette page à la soumission :

The page has expired due to inactivity.

Please refresh and try again.

Comme le token n'est pas bon, Laravel en conclut qu'il a expiré parce qu'évidemment il a une durée de validité limitée, liée à la session.

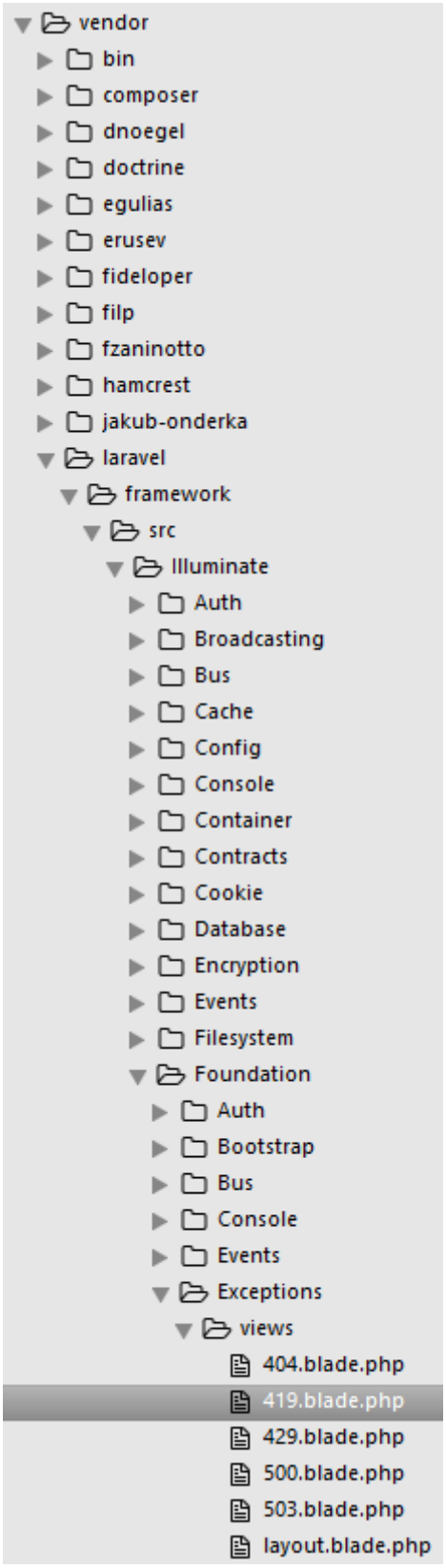
Analysons un peu mieux cette réponse :



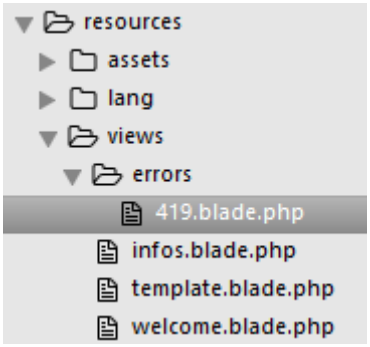
On voit une réponse 419 qui ne fait pas encore partie du standard HTTP mais qui est de plus en plus utilisée comme alternative à 401.

VII-G - Page d'erreur personnalisée

La page d'erreur sur laquelle on est tombé est explicite mais… en anglais. On pourrait la vouloir en français, ou tout simplement en changer le style. Mais où se trouve le code de cette page ? Regardez dans le dossier **vendor** :



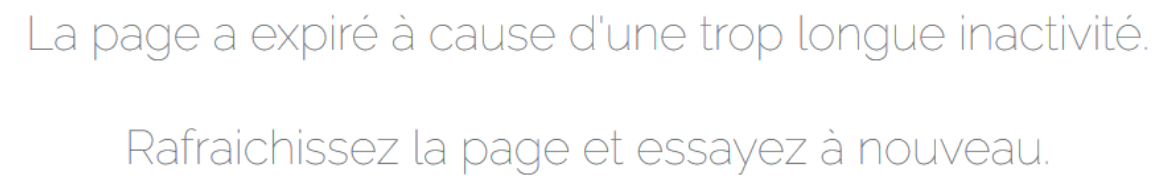
Vous trouvez un dossier des vues par défaut de Laravel pour les erreurs les plus classiques. Vous n'allez évidemment pas modifier directement ces vues dans le dossier **vendor** ! Mais vous pouvez surcharger ces vues en créant un dossier **resources/views/errors** et en copiant le fichier concerné :



Adaptez le code selon vos goûts :

```
1. @extends('errors::layout')
2.
3. @section('title', 'Page Expired')
4.
5. @section('message')
6.     La page a expiré à cause d'une trop longue inactivité.
7.     <br/><br/>
8.     Rafraîchissez la page et essayez à nouveau.
9. @stop
```

Et maintenant vous avez la page en français :



Vous pouvez procéder de la même manière avec tous les codes d'erreur.

VII-H - En résumé

- Laravel permet de créer des routes avec différents verbes : get, post…
- Un middleware permet de filtrer les requêtes.
- Les entrées du client sont récupérées dans la requête.
- On peut se prémunir contre les attaques CSRF.
- On peut personnaliser les pages d'erreur par défaut de Laravel et même en créer de nouvelles.

VIII - La validation

Nous avons vu dans le chapitre précédent un scénario mettant en œuvre un formulaire. Nous n'avons imposé aucune contrainte sur les valeurs transmises. Dans une application réelle, il est toujours nécessaire de vérifier que ces valeurs correspondent à ce qu'on attend. Par exemple un nom doit comporter uniquement des caractères alphabétiques et avoir une longueur maximale, une adresse email doit correspondre à un certain format.

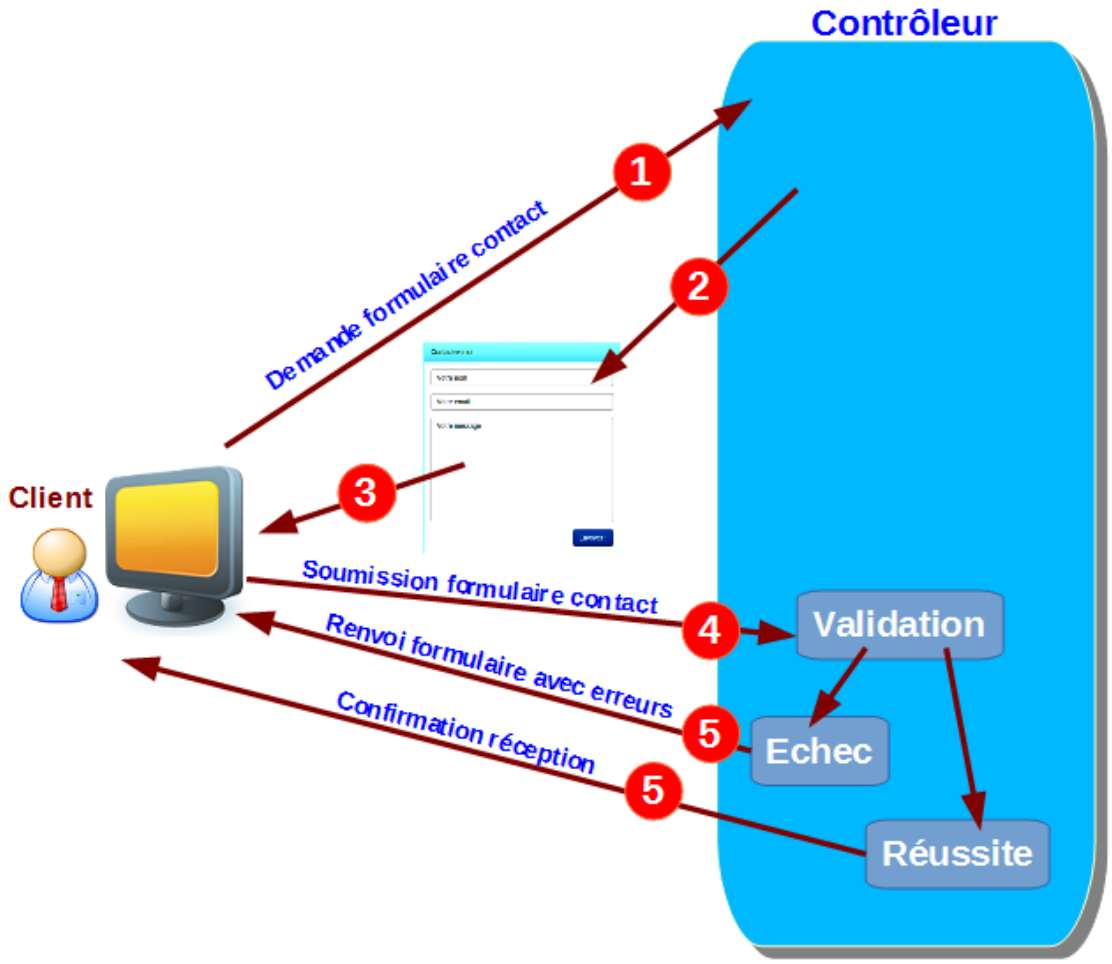
Il faut donc mettre en place des règles de validation. En général on procède à une première validation côté client pour éviter de faire des allers-retours avec le serveur. Mais quelle que soit la pertinence de cette validation côté client elle n'exonère pas d'une validation côté serveur.

On ne doit jamais faire confiance à des données qui arrivent sur le serveur !

Dans l'exemple de ce chapitre je ne prévoirai pas de validation côté client, d'une part ce n'est pas mon propos, d'autre part elle masquerait la validation côté serveur pour les tests.

VIII-A - Scénario et routes

Voici le scénario que je vous propose pour ce chapitre :



1 Le client demande le formulaire de contact ;
2 Le contrôleur génère le formulaire ;
3 Le contrôleur envoie le formulaire ;
4 Le client remplit le formulaire et le soumet ;
5 Le contrôleur teste la validité des informations et là on a deux possibilités :

1 en cas d'échec on renvoie le formulaire au client en l'informant des erreurs et en conservant ses entrées correctes ;
2 en cas de réussite on envoie un message de confirmation au client.

VIII-A-1 - Routes

On va donc avoir besoin de deux routes :

```
Route::get('contact', 'ContactController@create');
Route::post('contact', 'ContactController@store');
```

On aura une seule URL (avec verbe « get » pour demander le formulaire et verbe « post » pour le soumettre) :

```
http://monsite.fr/contact
```

VIII-B - Les vues

VIII-B-1 - Le template

Pour ce chapitre je vais créer un template réaliste avec l'utilisation de Bootstrap pour alléger le code. Voici le code de ce template (**resources/views/template.blade.php**) :

```
1. <!DOCTYPE html>
2. <html lang="fr">
3.     <head>
4.         <meta charset="utf-8">
5.         <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
6.         <title>Mon joli site</title>
7.
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css" integr
8.         <style>
9.             textarea { resize: none; }
10.             .card { width: 25em; }
11.         </style>
12.     </head>
13.     <body>
14.         @yield('contenu')
15.     </body>
16. </html>
```

J'ai prévu l'emplacement **@yield** nommé « contenu » pour recevoir les pages du site ; pour notre exemple on aura seulement la page de contact et celle de la confirmation.

VIII-B-2 - La vue de contact

La vue de contact va contenir essentiellement un formulaire (**resources/views/contact.blade.php**) :

```
1. @extends('template')
2.
3. @section('contenu')
4.     <br>
5.     <div class="container">
6.         <div class="row card text-white bg-dark">
7.             <h4 class="card-header">Contactez-moi</h4>
8.             <div class="card-body">
9.                 <form action="{{ url('contact') }}" method="POST">
10.                     {{ csrf_field() }}
11.                     <div class="form-group">
12.
  <input type="text" class="form-control {{ $errors->has('nom') ? 'is-invalid' : '' }}" name="nom" id="nom" place
 old('nom') }}">
13.
  {!! $errors->first('nom', '<div class="invalid-feedback">:message</div>') !!}
14.                     </div>
15.                     <div class="form-group">
16.
  <input type="email" class="form-control {{ $errors->has('email') ? 'is-invalid' : '' }}" name="email" id="email
 old('email') }}">
17.
  {!! $errors->first('email', '<div class="invalid-feedback">:message</div>') !!}
18.                     </div>
19.                     <div class="form-group">
20.
  <textarea class="form-control {{ $errors->has('message') ? 'is-invalid' : '' }}" name="message" id="message" pl
 old('message') }}</textarea>
```

```
21.
  {!! $errors->first('message', '<div class="invalid-feedback">:message</div>') !!}
22.                     </div>
23.                     <button type="submit" class="btn btn-secondary">Envoyer !</button>
24.                 </form>
25.             </div>
26.         </div>
27.     </div>
28. @endsection
```

Cette vue étend le template vu ci-dessus et renseigne la section « contenu ». Je ne commente pas la mise en forme spécifique à Bootstrap.

En cas de réception du formulaire suite à des erreurs on reçoit une variable **$errors** qui contient un tableau avec comme clés les noms des contrôles et comme valeurs les textes identifiant les erreurs.

*La variable **$errors** est générée systématiquement pour toutes les vues.*

C'est pour cela que je teste la présence d'une erreur pour chaque contrôle en ajustant le style et en affichant le texte de l'erreur si nécessaire avec la méthode **first** :

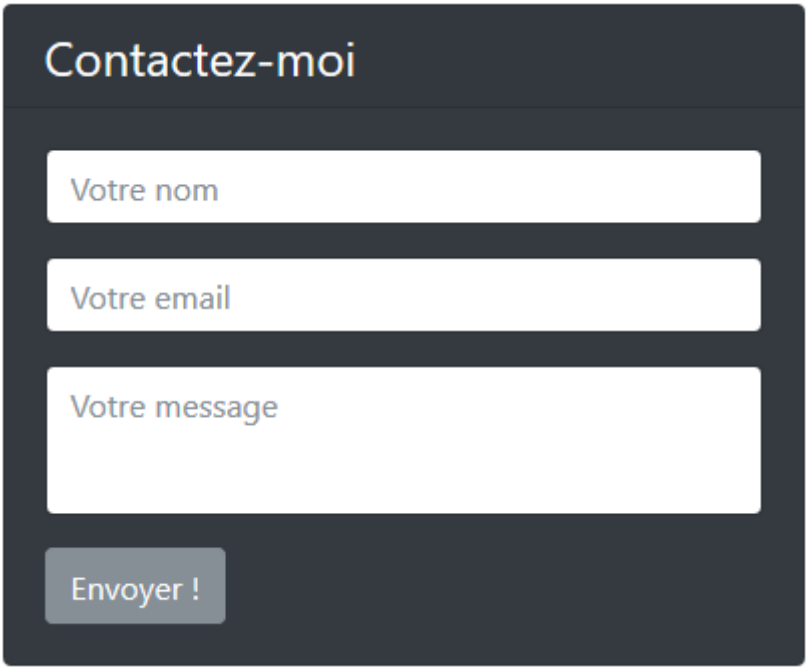
```
{!! $errors->first('nom', '<div class="invalid-feedback">:message</div>') !!}
```

S'il n'y a aucune erreur rien n'est renvoyé et donc rien n'est affiché, sinon on récupère la première (**first**) et on respecte le format imposé.

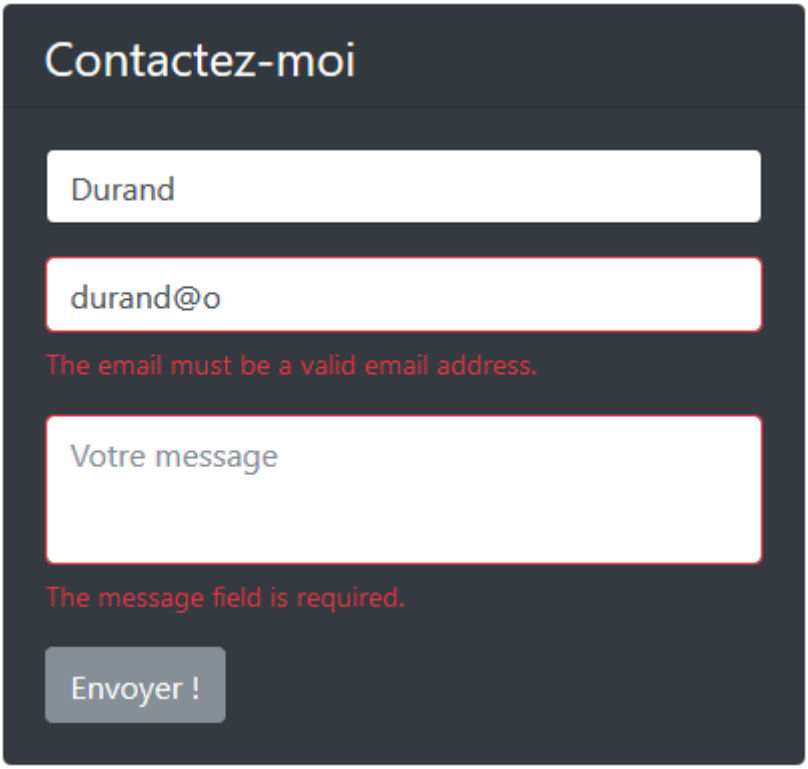
Enfin, en cas d'erreur de validation, les anciennes valeurs saisies sont retournées au formulaire et récupérées avec l'helper **old** :

```
value="{{ old('nom') }}"
```

Au départ le formulaire se présente ainsi :

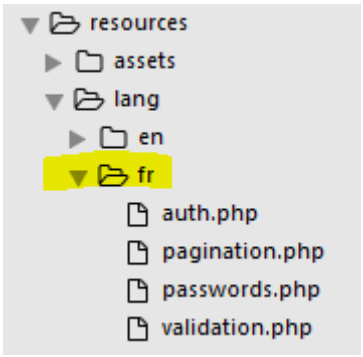


Après une soumission et renvoi avec des erreurs il pourra se présenter ainsi :



VIII-B-3 - Les messages en français

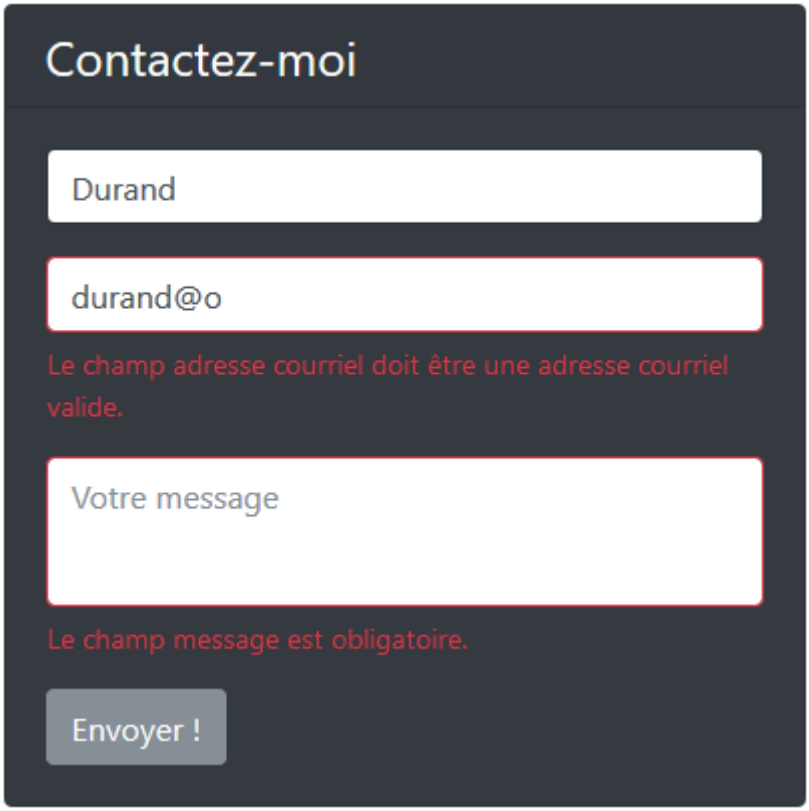
Par défaut les messages sont en anglais. Pour avoir ces textes en français vous devez récupérer les fichiers **ici**. Placez le dossier « fr » et son contenu dans le dossier **resources/lang** :



Ensuite changez cette ligne dans le fichier **config/app.php** :

```
'locale' => 'fr',
```

Vous devriez avoir vos messages en français :

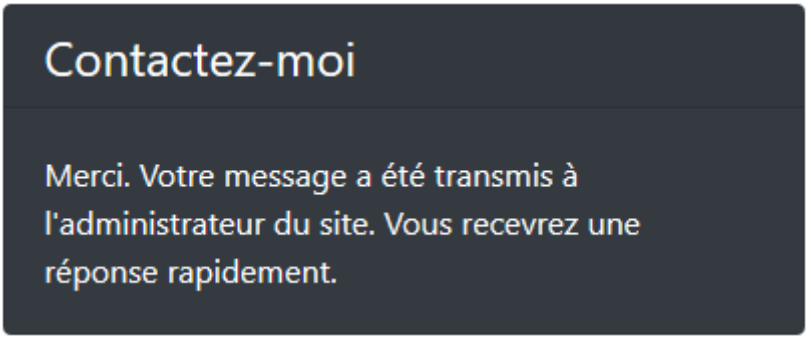


VIII-B-4 - La vue de confirmation

Pour la vue de confirmation (**resources/views/confirm.blade.php**) le code est plus simple et on utilise évidemment le même template :

```
1. @extends('template')
2.
3. @section('contenu')
4.     <br>
5.     <div class="container">
6.         <div class="row card text-white bg-dark">
7.             <h4 class="card-header">Contactez-moi</h4>
8.             <div class="card-body">
9.                 <p class="card-text">Merci. Votre message a été transmis à l'administrateur du
 site. Vous recevrez une réponse rapidement.</p>
10.             </div>
11.         </div>
12.     </div>
13. @endsection
```

Ce qui donne cette apparence :



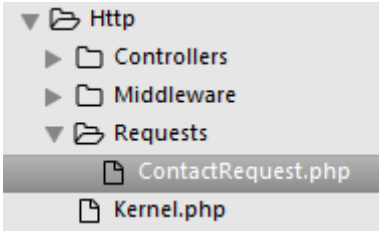
VIII-C - La requête de formulaire

Il y a plusieurs façons d'effectuer la validation avec Laravel mais la plus simple et élégante consiste à utiliser une requête de formulaire (**Form request**).

Nous avons déjà utilisé Artisan qui permet d'effectuer de nombreuses opérations et nous allons encore avoir besoin de lui pour créer une requête de formulaire :

```
php artisan make:request ContactRequest
```

Comme par défaut le dossier n'existe pas il est créé en même temps que la classe :



Voyons le code généré :

```php
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ContactRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
        ];
```

```
    }
}
```

La classe générée comporte deux méthodes :

- **authorize** : pour effectuer un contrôle de sécurité éventuel sur l'identité ou les droits de l'émetteur ;
- **rules** : pour les règles de validation.

On va arranger le code pour notre cas :

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ContactRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'nom' => 'bail|required|between:5,20|alpha',
            'email' => 'bail|required|email',
            'message' => 'bail|required|max:250'
        ];
    }
}
```

Au niveau de la méthode **rules** on retourne un tableau qui contient des clés qui correspondent aux champs du formulaire. Vous retrouvez le nom, l'email et le message. Les valeurs contiennent les règles de validation. Comme il y en a chaque fois plusieurs, elles sont séparées par le signe « | ». Voyons les différentes règles prévues :

- **bail** : on arrête de vérifier dès qu'une règle n'est pas respectée ;
- **required** : une valeur est requise, donc le champ ne doit pas être vide ;
- **between** : nombre de caractères entre une valeur minimale et une valeur maximale ;
- **alpha** : on n'accepte que les caractères alphabétiques ;
- **email** : la valeur doit être une adresse email valide.

Au niveau de la méthode **authorize** je me suis contenté de renvoyer **true** parce que nous ne ferons pas de contrôle supplémentaire à ce niveau.

> *Vous pouvez trouver toutes les règles disponibles* **dans la documentation**. *Vous verrez que la liste est longue !*

VIII-D - Le contrôleur

On va encore utiliser Artisan pour générer le contrôleur :

```
php artisan make:controller ContactController
```

Modifiez le code par défaut pour en arriver à celui-ci :

```
1. <?php
2.
3. namespace App\Http\Controllers;
4.
5. use App\Http\Requests\ContactRequest;
6.
7. class ContactController extends Controller
8. {
9.     public function create()
10.     {
11.         return view('contact');
12.     }
13.
14.     public function store(ContactRequest $request)
15.     {
16.         return view('confirm');
17.     }
18. }
```

La méthode **create** ne présente aucune nouveauté par rapport à ce qu'on a vu au chapitre précédent. On se contente de renvoyer la vue **contact** qui comporte le formulaire.

La méthode **store** nécessite quelques commentaires. Vous remarquez le paramètre de type **ContactRequest**. On injecte dans la méthode une instance de la classe **ContactRequest** que l'on a précédemment créée. Laravel permet ce genre d'injection de dépendance au niveau d'une méthode. Je reviendrai en détail dans un prochain chapitre sur cette possibilité.

Si la validation échoue parce qu'une règle n'est pas respectée c'est la classe **ContactRequest** qui s'occupe de tout : elle renvoie le formulaire en complétant les contrôles qui étaient corrects et crée une variable **$errors** pour transmettre les messages d'erreurs qu'on utilise dans la vue. Vous n'avez rien d'autre à faire !

Vérifiez avec la commande **php artisan route:list** que tout est correct :

```
λ php artisan route:list
+--------+----------+---------+------+-----------------------------------------------+------------+
| Domain | Method   | URI     | Name | Action                                        | Middleware |
+--------+----------+---------+------+-----------------------------------------------+------------+
|        | GET|HEAD | contact |      | App\Http\Controllers\ContactController@create | web        |
|        | POST     | contact |      | App\Http\Controllers\ContactController@store  | web        |
+--------+----------+---------+------+-----------------------------------------------+------------+
```

On a bien nos deux routes avec l'URL correcte, le bon contrôleur avec les méthodes prévues et le middleware **web** appliqué aux deux routes.

Faites quelques essais avec des erreurs de saisie pour voir le fonctionnement.

VIII-E - D'autre façons d'effectuer la validation

Si vous n'appréciez pas les requêtes de formulaire et leur côté « magique » vous pouvez effectuer la validation directement dans le contrôleur avec la méthode **validate**. Voici le contrôleur modifié en conséquence :

```
1. <?php
2.
3. namespace App\Http\Controllers;
4.
5. use Illuminate\Http\Request;
6.
7. class ContactController extends Controller
```

```
{
    public function create()
    {
        return view('contact');
    }

    public function store(Request $request)
    {
        $this->validate($request, [
            'nom' => 'bail|required|between:5,20|alpha',
            'email' => 'bail|required|email',
            'message' => 'bail|required|max:250'
        ]);

        return view('confirm');
    }
}
```

Cette fois on injecte dans la méthode **store** directement la requête (**Illuminate\Http\Request**). Le fonctionnement est exactement le même.

Si cette méthode **validate** est encore trop abstraite à votre goût vous pouvez détailler les opérations :

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Validator;

class ContactController extends Controller
{
    public function create()
    {
        return view('contact');
    }

    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'nom' => 'bail|required|between:5,20|alpha',
            'email' => 'bail|required|email',
            'message' => 'bail|required|max:250'
        ]);

        if ($validator->fails()) {
            return back()->withErrors($validator)->withInput();
        }

        return view('confirm');
    }
}
```

On utilise la façade **Validator** en précisant toutes les entrées (**$request->all()**) et les règles de validation. Ensuite si la validation échoue (**fails**) on renvoie le formulaire (**back**) avec les erreurs (**withErrors**) et les valeurs entrées (**withInput**) pour pouvoir les afficher dans le formulaire.

Mais pourquoi se compliquer la vie quand on dispose de fonctionnalités plus simples et élégantes ?

VIII-F - En résumé

- La validation est une étape essentielle de vérification des entrées du client.
- On dispose de nombreuses règles de validation.
- Le validateur génère des erreurs explicites à afficher au client.

- Pour avoir les textes des erreurs en français il faut aller chercher les traductions et les placer dans le bon dossier.
- Les requêtes de formulaires (**Form request**) permettent d'effectuer la validation de façon simple et élégante.
- Il y a plusieurs façons d'effectuer la validation à adapter selon les goûts et les circonstances.

IX - Envoyer un email

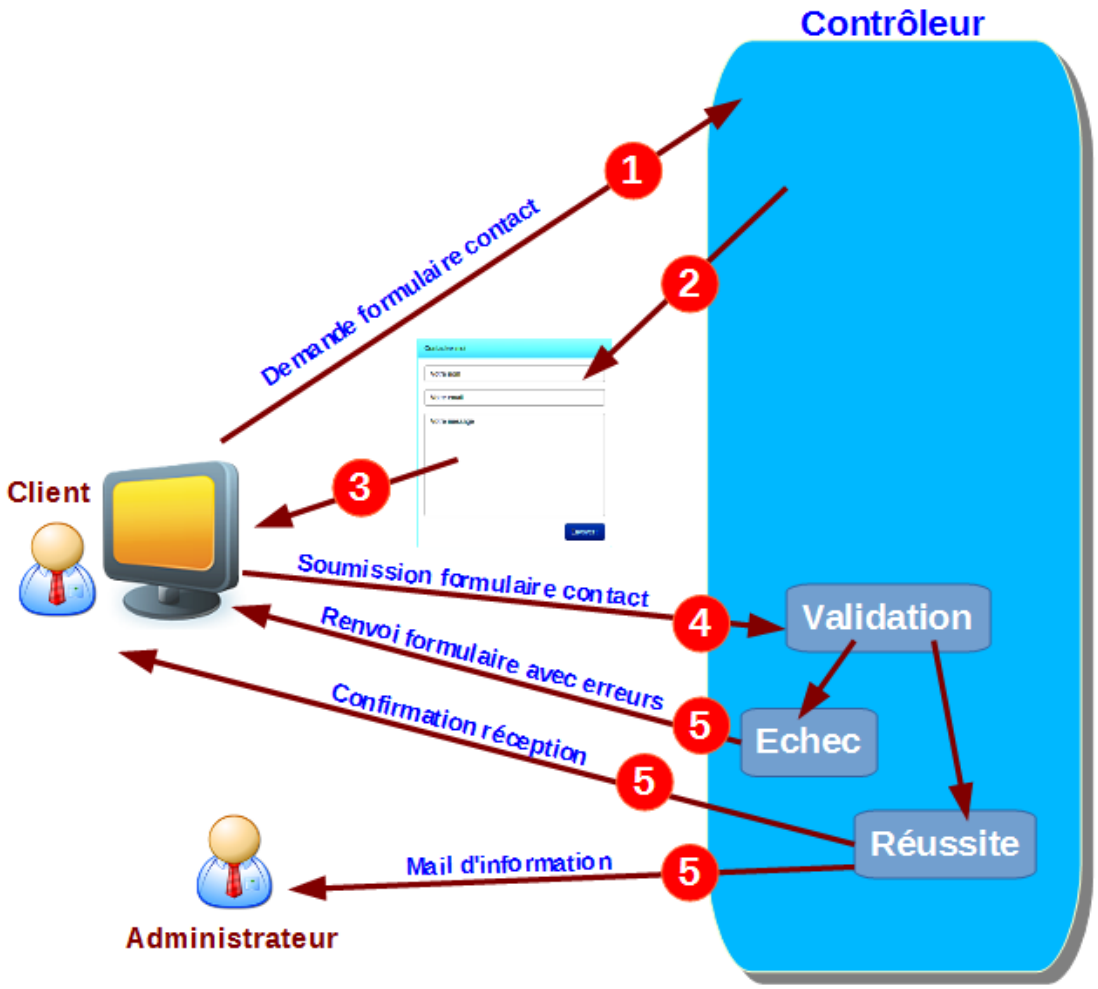
Laravel utilise le célèbre composant **SwiftMailer** pour l'envoi des emails. Mais il en simplifie grandement l'utilisation. Dans ce chapitre nous allons prolonger l'exemple précédent de la prise de contact en ajoutant l'envoi d'un email à l'administrateur du site lorsque quelqu'un soumet une demande de contact.

On va donc prendre le code tel qu'on l'a laissé lors du précédent chapitre et le compléter en conséquence.

On verra plus tard que Laravel propose aussi un système complet de notification qui permet entre autres l'envoi d'emails.

IX-A - Le scénario

Le scénario est donc le même que pour le précédent chapitre avec l'ajout d'une action :



On va avoir les mêmes routes et vues, c'est uniquement au niveau du contrôleur que le code va évoluer pour intégrer cette action complémentaire.

IX-B - Configuration

Si vous regardez dans le fichier **.env** vous trouvez une section qui concerne les emails :

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

Les valeurs correspondent au prestataire utilisé.

Au niveau du driver, le plus classique est certainement le SMTP mais vous pouvez aussi utiliser **mail**, **mailgun** (gratuit jusqu'à 10 000 envois par mois), **ses**, **sparkpost**…

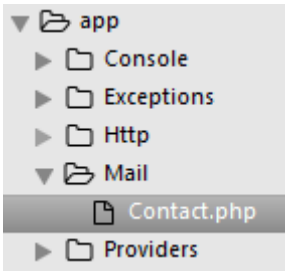
Vous devez correctement renseigner les paramètres pour que ça fonctionne selon votre contexte (en local avec le SMTP de votre prestataire, en production avec la fonction mail de PHP ou d'un autre système…).

IX-C - La classe Mailable

Avec Laravel, pour envoyer un email il faut passer par la création d'une classe « mailable ». Encore une fois c'est Artisan qui va nous permettre de créer notre classe :

```
php artisan make:mail Contact
```

Comme le dossier n'existe pas il est créé en même temps que le fichier de la classe :



Par défaut on a ce code :

```php
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class Contact extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct()
    {
```

```
        //
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('view.name');
    }
}
```

Tout se passe dans la méthode **build**. On voit qu'on retourne une vue ; celle-ci doit comporter le code pour le contenu de l'email.

On commence par changer le nom de la vue :

```
return $this->view('emails.contact');
```

On va créer cette vue (**resources/views/emails/contact.blade.php**) :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <h2>Prise de contact sur mon beau site</h2>
    <p>Réception d'une prise de contact avec les éléments suivants :</p>
    <ul>
       <li><strong>Nom</strong> : {{ $contact['nom'] }}</li>
       <li><strong>Email</strong> : {{ $contact['email'] }}</li>
       <li><strong>Message</strong> : {{ $contact['message'] }}</li>
     </ul>
  </body>
</html>
```

Pour que ça fonctionne on doit transmettre à cette vue les entrées de l'utilisateur. Il faut donc passer les informations à la classe **Contact** à partir du contrôleur.

On peut aussi **créer des email en Markdown**.

IX-D - Transmission des informations à la vue

Il y a deux façons de procéder pour transmettre les informations, nous allons utiliser la plus simple. Il suffit de créer une propriété obligatoirement publique dans la classe « mailable » et celle-ci sera automatiquement transmise à la vue. Voici le nouveau code de notre classe :

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class Contact extends Mailable
{
    use Queueable, SerializesModels;

    /**
```

```
15.       * Elements de contact
16.       * @var array
17.       */
18.      public $contact;
19.
20.      /**
21.       * Create a new message instance.
22.       *
23.       * @return void
24.       */
25.      public function __construct(Array $contact)
26.      {
27.          $this->contact = $contact;
28.      }
29.
30.      /**
31.       * Build the message.
32.       *
33.       * @return $this
34.       */
35.      public function build()
36.      {
37.          return $this->from('monsite@chezmoi.com')
38.              ->view('emails.contact');
39.      }
40. }
```

J'en ai aussi profité pour préciser l'adresse de l'expéditeur avec la méthode **from**. On pourrait attacher un document avec **attach**, faire une copie avec **cc**…

J'ai créé la propriété publique **$contact** qui sera renseignée par l'intermédiaire du constructeur.

Il ne reste plus qu'à modifier le contrôleur pour envoyer cet email avec les données nécessaires :

```
1. <?php
2.
3. namespace App\Http\Controllers;
4.
5. use App\Http\Requests\ContactRequest;
6. use Illuminate\Support\Facades\Mail;
7. use App\Mail\Contact;
8.
9. class ContactController extends Controller
10. {
11.      public function create()
12.      {
13.          return view('contact');
14.      }
15.
16.      public function store(ContactRequest $request)
17.      {
18.          Mail::to('administrateur@chezmoi.com')
19.              ->send(new Contact($request->except('_token')));
20.
21.          return view('confirm');
22.      }
23. }
```

Tout se passe sur cette ligne :

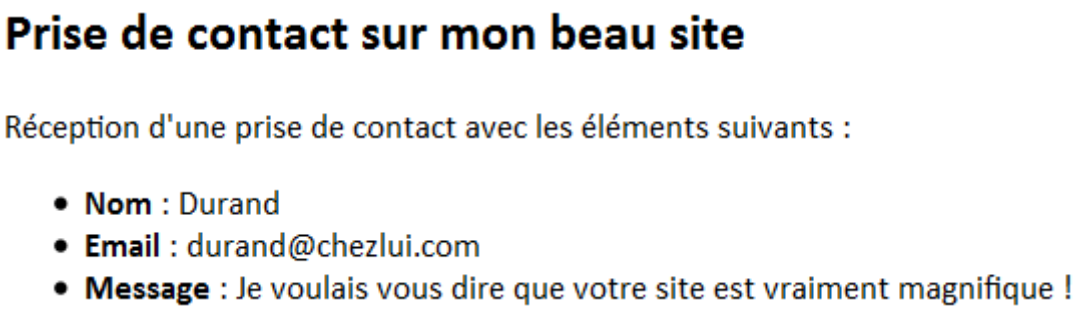
```
Mail::to('administrateur@chezmoi.com')
    ->send(new Contact($request->except('_token')));
```

On a :

- l'adresse de l'administrateur (**to**) ;

- l'envoi avec la méthode **send** ;
- la création d'une instance de la classe **Contact** avec la transmission des données saisies (mis à part le token pour la protection CSRF qui ne sert pas).

Et si tout se passe bien le message doit arriver jusqu'à l'administrateur :



Comme l'envoi d'emails peut prendre beaucoup de temps on utilise en général une file d'attente (**queue**). Il faut changer ainsi le code dans le contrôleur :

```
Mail::to('administrateur@chezmoi.com')
            ->queue(new Contact($request->except('_token')));
```

Mais évidemment pour que ça fonctionne il faut avoir paramétré et lancé **un système de file d'attente**. J'en parlerai sans doute dans un chapitre ultérieur.

IX-D-1 - Test de l'email

Quand on crée la vue pour l'email il est intéressant de voir l'aspect final avant de faire un envoi réel. Pour le faire il suffit de créer une simple route :

```
1. Route::get('/test-contact', function () {
2.     return new App\Mail\Contact([
3.         'nom' => 'Durand',
4.         'email' => 'durand@chezlui.com',
5.         'message' => 'Je voulais vous dire que votre site est magnifique !'
6.         ]);
7. });
```

Maintenant en utilisant l'URL **monsite/test-contact** on obtient l'aperçu directement dans le navigateur !

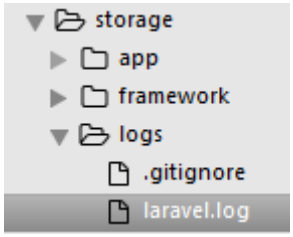
IX-E - Envoyer des emails en phase développement

IX-E-1 - Le mode Log

Lorsqu'on est en phase de développement il n'est pas forcément pratique ou judicieux d'envoyer réellement des emails. Une solution simple consiste à passer en mode Log en renseignant le driver dans le fichier **.env** :

```
MAIL_DRIVER=log
```

Les emails ne seront pas envoyés mais le code en sera mémorisé dans le fichier des logs :



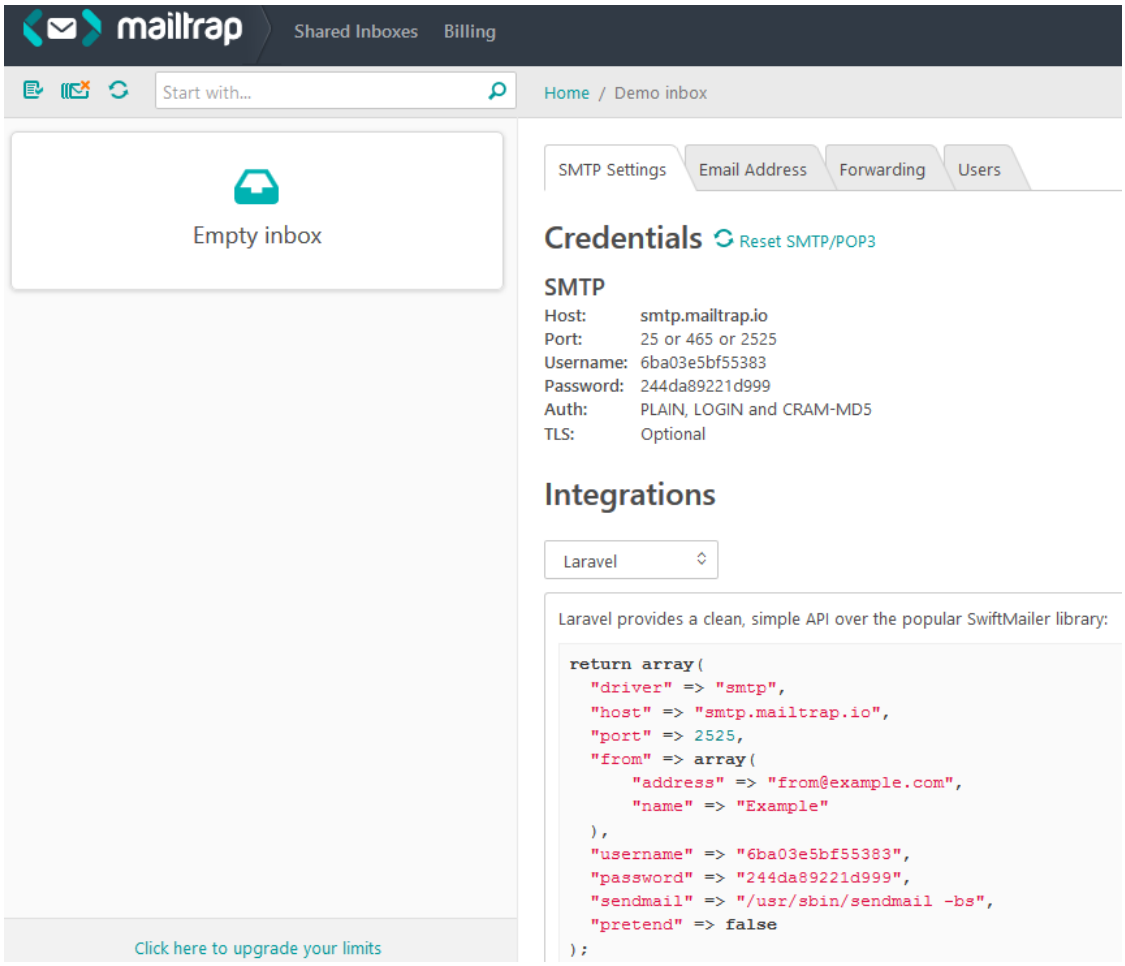
Vous allez y trouver par exemple ce contenu :

```
[2017-09-10 14:18:13] local.DEBUG: Message-ID: <b25ad1d634f7ca660d1d7bbf81a8463a@laravel5.dev>
Date: Sun, 10 Sep 2017 14:18:13 +0000
Subject: Contact
From: monsite@chezmoi.com
To: administrateur@chezmoi.com
MIME-Version: 1.0
Content-Type: text/html; charset=utf-8
Content-Transfer-Encoding: quoted-printable

<!DOCTYPE html>
<html lang="fr">
 <head>
 <meta charset="utf-8">
 </head>
 <body>
 <h2>Prise de contact sur mon beau site</h2>
 <p>Réception d'une prise de contact avec les éléments suivants :</p>
 <ul>
 <li><strong>Nom</strong> : Durand</li>
 <li><strong>Email</strong> : durand@chezlui.fr</li>
 <li><strong>Message</strong> : Je voulais vous dire que votre site est magnifique !</li>
 </ul>
 </body>
</html>
```

Ce qui vous permet de vérifier que tout se passe correctement (hormis l'envoi).

IX-E-2 - MailTrap

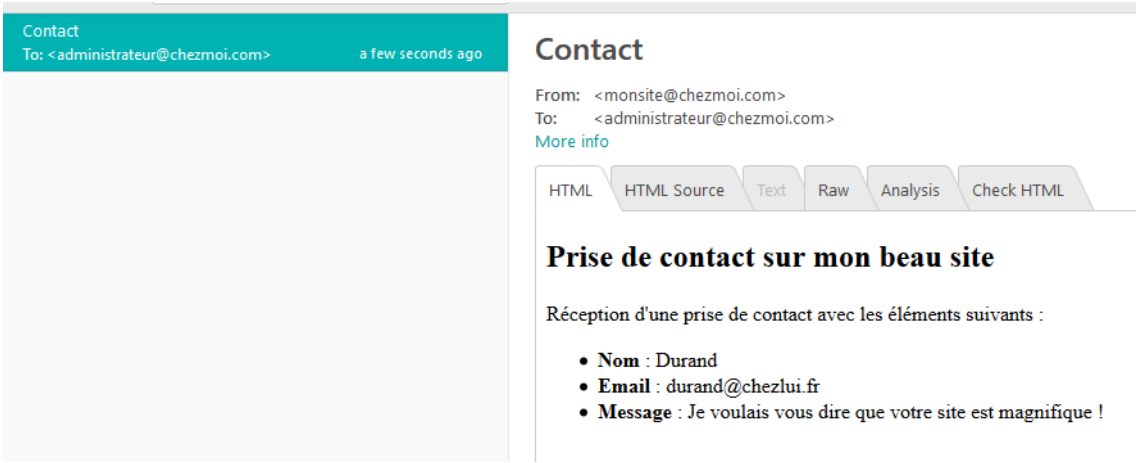
Une autre possibilité très utilisée est MailTrap qui a une option gratuite (une boîte, avec 50 messages au maximum et deux messages par seconde). Vous avez un tableau de bord et une boîte de messages :



J'ai fait apparaître les configurations pour Laravel. Il est ainsi facile de renseigner le fichier **.env** :

```
1. MAIL_DRIVER=smtp
2. MAIL_HOST=smtp.mailtrap.io
3. MAIL_PORT=2525
4. MAIL_USERNAME=6ba03e5bf55383
5. MAIL_PASSWORD=244da89221d999
6. MAIL_ENCRYPTION=null
```

Avec cette configuration lorsque j'envoie un email je le vois arriver :



Vous pouvez analyser l'email avec précision, je vous laisse découvrir toutes les options.

IX-F - En résumé

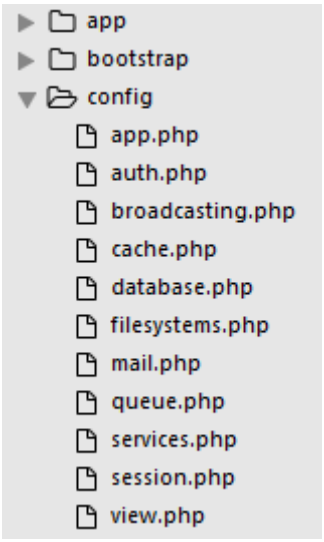
- Laravel permet l'envoi simple d'email.
- Il faut configurer correctement les paramètres pour l'envoi des emails.
- Pour chaque email il faut créer une classe « mailable ».
- On peut passer des informations à l'email en créant une propriété publique dans la classe « mailable ».
- On peut passer en mode Log en phase de développement ou alors utiliser MailTrap.

X - Configuration, session et gestion de fichiers

Dans ce chapitre nous verrons la configuration, la gestion des sessions et des fichiers avec un exemple simple d'envoi et d'enregistrement de fichiers images dans un dossier à partir d'un formulaire.

X-A - La configuration

Tout ce qui concerne la configuration de Laravel se trouve dans le dossier **config** :



De nombreuses valeurs de configuration sont définies dans le fichier **.env**.

Les fichiers de configuration contiennent en fait juste un tableau avec des clés et des valeurs. Par exemple pour les vues (**view.php**) :

```
1. <?php
2.
3. return [
4.
5.     'paths' => [
6.         resource_path('views'),
7.     ],
8.
9.     ...
10.
11. ];
```

On a la clé **paths** et la valeur : un tableau des clés et des valeurs. Pour récupérer une valeur il suffit d'utiliser sa clé avec l'helper **config** :

```
config('view.paths');
```

On utilise le nom du fichier (**view**) et le nom de la clé (**paths**) séparés par un point.

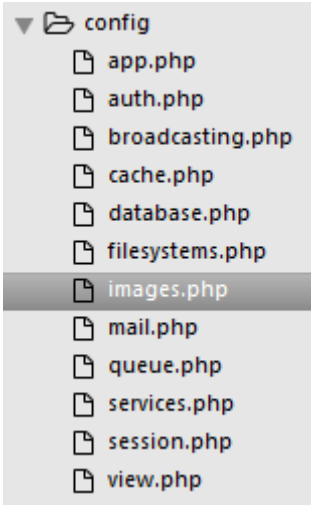
On peut aussi changer une valeur :

```
config(['view.paths' => resource_path().'/mes_vues']);
```

Si je fais effectivement cela, mes vues, au lieu d'être cherchées dans le dossier **resources/views** seront cherchées dans le dossier **resources/mes_vues**.

> *Lorsqu'on change ainsi une valeur de configuration ce n'est valable que pour la requête en cours.*

Vous pouvez évidemment créer vos propres fichiers de configuration. Pour l'exemple de ce chapitre on va avoir besoin justement d'utiliser une configuration. Comme notre application doit enregistrer des fichiers d'images dans un dossier il faut définir l'emplacement et le nom de ce dossier de destination. On va donc créer un fichier **images.php** :



Dans ce fichier on va définir le nom du dossier :

```
return ['path' => 'uploads'];
```

*En production pour gagner en performance il est conseillé de mettre toute la configuration en cache dans un seul fichier avec la commande **php artisan config:cache**.*

X-B - Les sessions

Étant donné que les requêtes HTTP sont fugitives et ne laissent aucune trace, il est important de disposer d'un système qui permet de mémoriser des informations entre deux requêtes. C'est justement l'objet des sessions.

La configuration des sessions se trouve dans le fichier de configuration **session.php**. On trouve dans le fichier **.env** la définition du driver :

```
SESSION_DRIVER=file
```

Par défaut c'est un fichier (dans **storage/framework/sessions**) qui mémorise les informations des sessions mais on peut aussi utiliser : les cookies, la base de données, **apc**, **memcached**, **redis**…

Quel que soit le driver utilisé l'helper **session** de Laravel permet une gestion simplifiée des sessions. Vous pouvez ainsi créer une variable de session :

```
session(['clé' => 'valeur']);
```

Vous pouvez aussi récupérer une valeur à partir de sa clé :

```
$valeur = session('clef');
```

Vous pouvez prévoir une valeur par défaut :

```
$valeur = session('clef', 'valeur par défaut');
```

Il est souvent utile (ce sera le cas pour notre exemple) de savoir si une certaine clé est présente en session :

```
if (session()->has('error')) ...
```

Ces informations demeurent pour le même client à travers ses requêtes. Laravel s'occupe de ces informations, on se contente de lui indiquer un couple clé-valeur et il s'occupe de tout.

Ce ne sont là que les méthodes de base pour les sessions, vous trouverez tous les renseignements complémentaires **dans la documentation**.

X-C - La gestion des fichiers

Laravel utilise **Flysystem** comme composant de gestion de fichiers. Il en propose une API bien pensée et facile à utiliser. On peut ainsi manipuler fichiers et dossiers de la même manière en local ou à distance, que ce soit en FTP ou sur le cloud.

La configuration du système se trouve dans le fichier **config/filesystem.php**. Par défaut on est en local :

```
'default' => env('FILESYSTEM_DRIVER', 'local'),
```

Mais on peut aussi choisir ftp, **s3** ou **rackspace** (pour ces deux derniers il faut installer les composants correspondants). On peut aussi mettre en place un accès à d'autres possibilités et il existe des composants à ajouter, comme par exemple **celui-ci pour Dropbox**.

On peut définir des « disques » (**disks**) qui sont des cibles combinant un driver, un dossier racine et différents éléments de configuration :

```
1. 'disks' => [
2.
3.     'local' => [
4.         'driver' => 'local',
5.         'root' => storage_path('app'),
6.     ],
7.
8.     'public' => [
9.         'driver' => 'local',
10.         'root' => storage_path('app/public'),
11.         'url' => env('APP_URL').'/storage',
12.         'visibility' => 'public',
13.     ],
14.
15.     's3' => [
16.         'driver' => 's3',
17.         'key' => env('AWS_KEY'),
18.         'secret' => env('AWS_SECRET'),
19.         'region' => env('AWS_REGION'),
20.         'bucket' => env('AWS_BUCKET'),
21.     ],
22.
23. ],
```

Donc par défaut c'est le disque local qui est actif et le dossier racine est **storage/app**.

Autrement dit si j'utilise ce code :

```
Storage::disk('local')->put('recettes.txt', 'Contenu du fichier');
```

Je vais envoyer le fichier **recettes.txt** dans le dossier **storage/app**.

Et si j'utilise ce code :

```
Storage::disk('public')->put('recettes.txt', 'Contenu du fichier');
```

Cette fois j'envoie le fichier dans le dossier **storage/app/public**. Par convention ce dossier doit être accessible depuis le web.

Mais je vous ai dit précédemment que seul le dossier **public** à la racine doit être accessible. Alors ?

Alors pour que ce soit possible il faut créer un lien symbolique **public/storage** qui pointe sur **storage/app/public**. Il y a d'ailleurs une commande d'Artisan pour ça :

```
php artisan storage:link
```

Mais franchement je préfère créer directement un dossier dans **public**. Les motivations avancées (ne pas perdre de fichiers au déploiement) me paraissent trop minces.

Donc rien n'empêche de changer la configuration :

```
1. 'public' => [
2.     'driver' => 'local',
3.     'root' => public_path(),
4.     'visibility' => 'public',
5. ],
```

Ainsi le disque « public » pointe sur le dossier **public**. C'est d'ailleurs ce qu'on va faire pour l'exemple de ce chapitre.

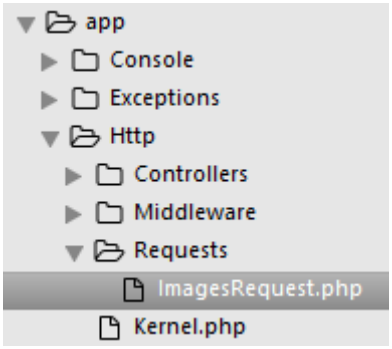
Il y a de nombreuses méthodes pour manipuler dossiers et fichiers, je ne vais pas développer tout ça pour le moment, vous avez le détail dans la documentation. On va surtout utiliser les possibilités de téléchargement des fichiers dans ce chapitre.

X-D - La requête de formulaire

Nous allons encore avoir besoin d'une requête de formulaire pour la validation. Comme nous l'avons déjà vu nous utilisons la commande d'Artisan pour la créer :

```
php artisan make:request ImagesRequest
```

Vous devez trouver le fichier ici :



Changez le code pour celui-ci :

```php
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ImagesRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return ['image' => 'required|image|dimensions:min_width=100,min_height=100'];
    }
}
```

On a seulement trois règles pour le champ **image** :

- le champ est obligatoire (**required**) ;
- ce doit être une image (**image**) ;
- l'image doit faire au minimum 100 * 100 pixels (**dimensions**).

Maintenant notre validation est prête.

X-E - Les routes et le contrôleur

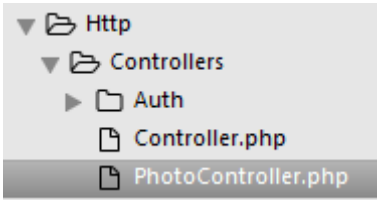
On va avoir besoin de deux routes :

```php
Route::get('photo', 'PhotoController@create');
Route::post('photo', 'PhotoController@store');
```

On utilise Artisan pour créer le contrôleur :

```
php artisan make:controller PhotoController
```

Vous devez trouver le fichier ici :



Changez le code pour celui-ci :

```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests\ImagesRequest;

class PhotoController extends Controller
{

    public function create()
    {
        return view('photo');
    }

    public function store(ImagesRequest $request)
    {
        $request->image->store(config('images.path'), 'public');

        return view('photo_ok');
    }
}
```

Donc au niveau des URL :

- **http://monsite.fr/photo** avec le verbe **get** pour la demande du formulaire ;
- **http://monsite.fr/photo** avec le verbe **post** pour la soumission du formulaire et l'envoi du fichier image associé.

En ce qui concerne le traitement de la soumission, vous remarquez qu'on récupère le chemin du dossier d'enregistrement qu'on a prévu dans la configuration :

```php
config('images.path')
```

La partie intéressante se situe avec ce code :

```php
$request->image->store(config('images.path'), 'public')
```

On récupère l'image avec **$request->image**. Ce qu'on obtient là est une instance de la classe **Illuminate/Http/UploadedFile**.

Laravel dispose d'une documentation complète de ses classes, par exemple vous trouvez **cette classe ici** avec **la méthode store documentée**.

La méthode **store** génère automatiquement un nom de fichier basé sur son contenu (hashage MD5) et elle retourne le chemin complet.

X-F - Les vues

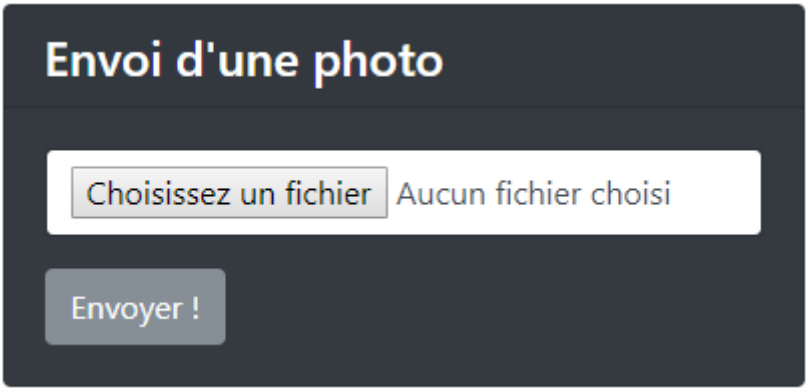
On va utiliser le template des chapitres précédents (**resources/views/template.blade.php**) :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
        <title>Mon joli site</title>

  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css" integr
        <style>
            textarea { resize: none; }
             .card { width: 25em; }
         </style>
    </head>
    <body>
        @yield('contenu')
    </body>
</html>
```

Voici la vue pour le formulaire (**resources/views/photo.blade.php**) :

```
@extends('template')

@section('contenu')
    <br>
    <div class="container">
        <div class="row card text-white bg-dark">
            <h4 class="card-header">Envoi d'une photo</h4>
            <div class="card-body">
                <form action="{{
 url('photo') }}" method="POST"  enctype="multipart/form-data">
                    {{ csrf_field() }}
                    <div class="form-group">

  <input type="file" class="form-control {{ $errors->has('image') ? 'is-invalid' : '' }}" name="image" id="image"
 old('image') }}">

  {!! $errors->first('image', '<div class="invalid-feedback">:message</div>') !!}
                    </div>
                    <button type="submit" class="btn btn-secondary">Envoyer !</button>
                </form>
            </div>
        </div>
    </div>
@endsection
```

Avec cet aspect :



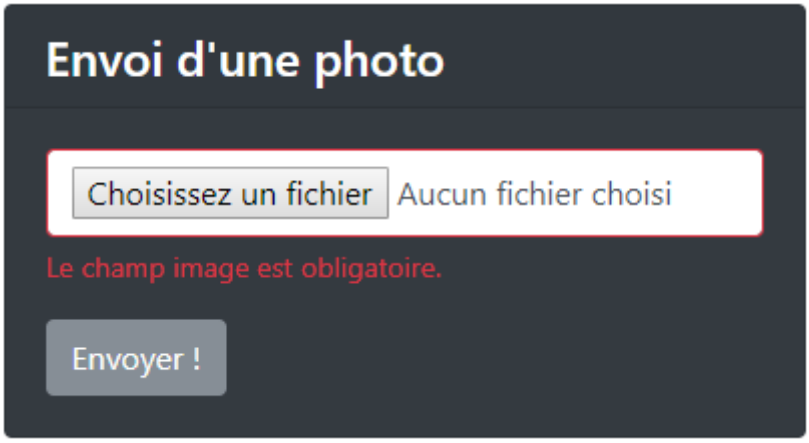
Remarquez comment est créé le formulaire :

```
<form action="{{ url('photo') }}" method="POST"  enctype="multipart/form-data">
```

On a prévu le type **mime** nécessaire pour associer un fichier lors de la soumission :

```
enctype="multipart/form-data"
```

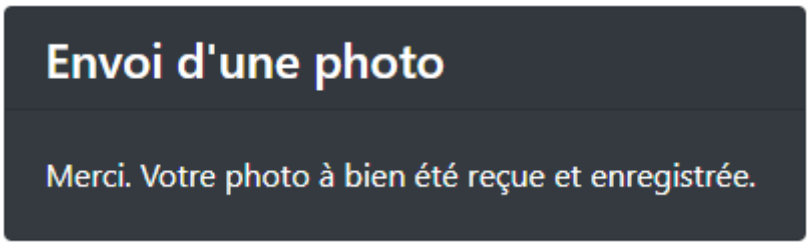
En cas d'erreur de validation le message est affiché et la bordure du champ devient rouge :



Et voici la vue pour la confirmation en retour (**app/views/photo_ok.blade.php**) :

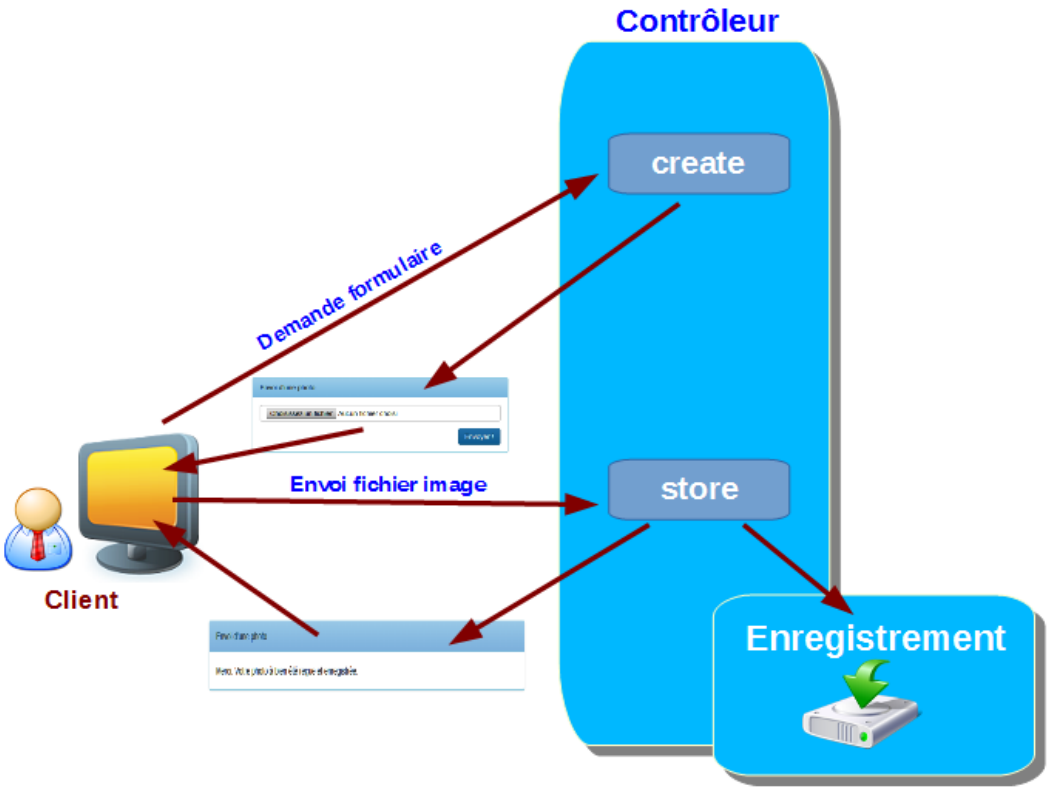
```
1. @extends('template')
2.
3. @section('contenu')
4.     <br>
5.     <div class="container">
6.         <div class="row card text-white bg-dark">
7.             <h4 class="card-header">Envoi d'une photo</h4>
8.             <div class="card-body">
9.                 <p class="card-text">Merci. Votre photo à bien été reçue et enregistrée.</p>
10.            </div>
11.        </div>
12.    </div>
13. @endsection
```

Avec cet aspect :



On retrouve normalement le fichier bien rangé dans le dossier prévu :

Voici le schéma de fonctionnement :



X-G - En résumé

- Les fichiers de configuration permettent de mémoriser facilement des ensembles clé-valeur et sont gérés par l'helper **config**.
- Les sessions permettent de mémoriser des informations concernant un client et sont facilement manipulables avec l'helper **session**.
- Laravel comporte un système complet de gestion de fichiers en local, à distance ou sur le cloud avec une API commune.
- Il est facile de créer un système de téléchargement de fichier.

XI - Injection de dépendance, conteneur et façades

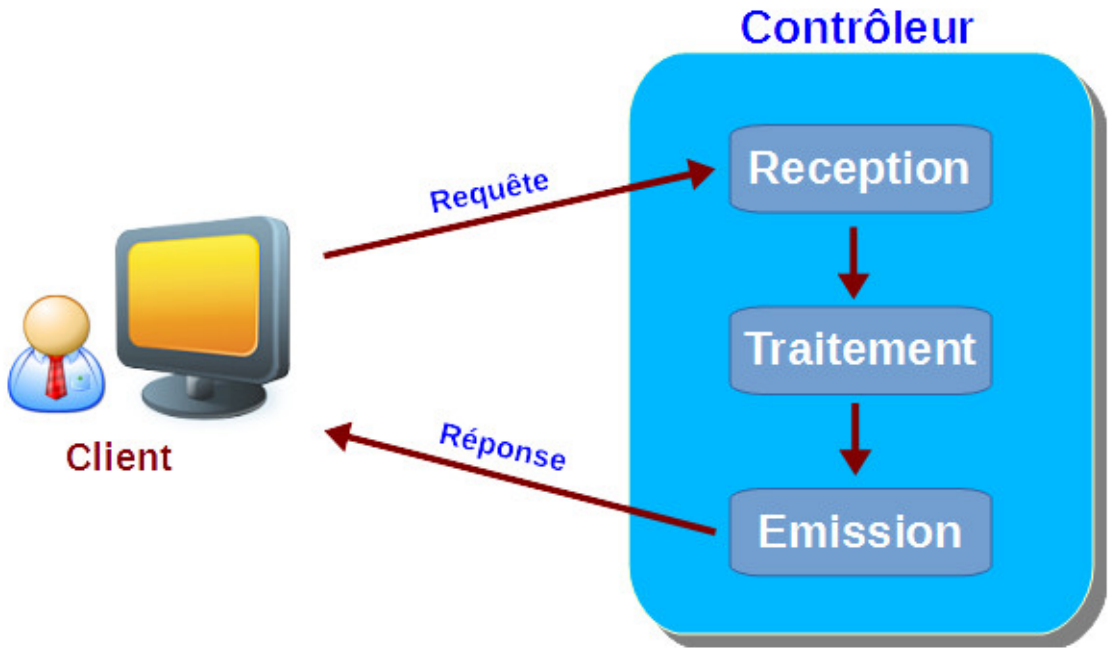
Dans ce chapitre nous allons reprendre l'exemple précédent de l'envoi de photos en nous posant des questions d'organisation du code. Laravel n'est pas seulement un framework pratique, c'est aussi un style de programmation. Il vaut mieux évoquer ce style le plus tôt possible dans l'apprentissage pour prendre rapidement les bonnes habitudes.

Vous pouvez très bien créer un site complet dans le fichier des routes, vous pouvez aussi vous contenter de contrôleurs pour effectuer tous les traitements nécessaires. Je vous propose une autre approche, plus en accord avec ce que nous offre Laravel.

XI-A - Le problème et sa solution

XI-A-1 - Le problème

Je vous ai déjà dit qu'un contrôleur a pour mission de réceptionner les requêtes et d'envoyer les réponses. Entre les deux il y a évidemment du traitement à effectuer, la réponse doit se construire, parfois c'est très simple, parfois plus long et délicat. Mais globalement nous avons pour un contrôleur ce fonctionnement :



Reprenons la méthode **store** de notre contrôleur **PhotoController** du précédent chapitre :

```
1. public function store(ImagesRequest $request)
2. {
3.     $request->image->store(config('images.path'), 'public');
4.
5.     return view('photo_ok');
6. }
```

Qu'avons-nous comme traitement ? On récupère l'image transmise et on enregistre cette image.

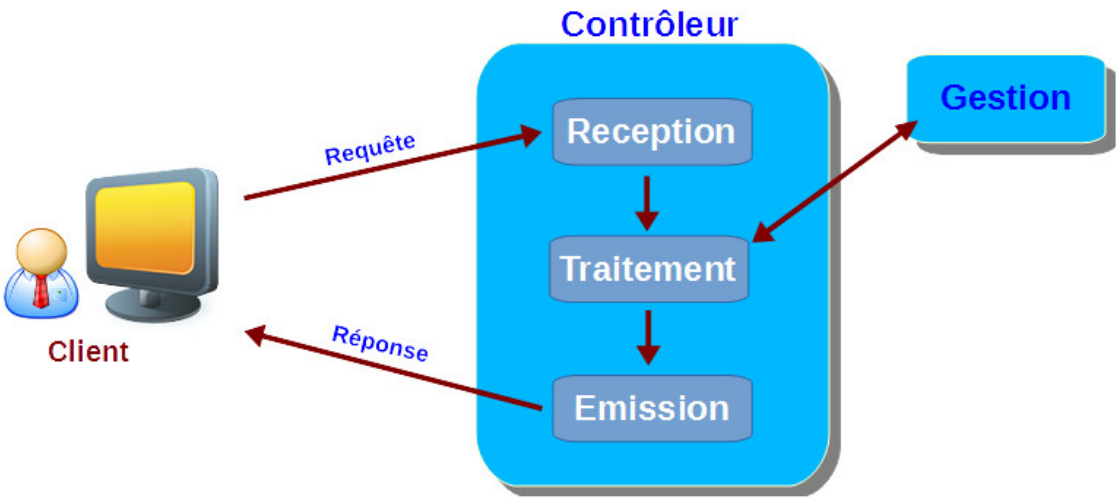
La question est : est-ce qu'un contrôleur doit savoir comment s'effectue ce traitement ? Si vous avez plusieurs contrôleurs dans votre application qui doivent effectuer le même traitement vous allez multiplier cette mise en place. Imaginez que vous ayez ensuite envie de modifier l'enregistrement des images, par exemple en les mettant sur le cloud, vous allez devoir retoucher le code de tous vos contrôleurs ! La répétition de code n'est jamais une bonne chose, une saine règle de programmation veut qu'on commence à se poser des questions sur l'organisation du code dès qu'on fait des copies.

Cette préconisation est connue sous l'acronyme DRY (« Don't Repeat Yourself »).

Un autre élément à prendre en compte aussi est la testabilité des classes. Nous verrons cet aspect important du développement trop souvent négligé. Pour qu'une classe soit testable il faut que sa mission soit simple et parfaitement identifiée et il ne faut pas qu'elle soit étroitement liée avec une autre classe. En effet cette dépendance rend les tests plus difficiles.

XI-A-2 - La solution

Alors quelle est la solution ? L'injection de dépendance ! Voyons de quoi il s'agit. Regardez ce schéma :



Une nouvelle classe entre en jeu pour la gestion, c'est elle qui est effectivement chargée du traitement, le contrôleur fait juste appel à ses méthodes. Mais comment cette classe est-elle injectée dans le contrôleur ? Voici le code du contrôleur modifié :

```
1. <?php
2.
3. namespace App\Http\Controllers;
4.
5. use App\Http\Requests\ImagesRequest;
6. use App\Repositories\PhotosRepository;
7.
8. class PhotoController extends Controller
9. {
10.     public function create()
11.     {
12.         return view('photo');
13.     }
14.
15.     public function store(ImagesRequest $request, PhotosRepository $photosRepository)
16.     {
17.         $photosRepository->save($request->image);
18.
19.         return view('photo_ok');
20.     }
21. }
```

Vous remarquez qu'au niveau de la méthode **store** il y a un nouveau paramètre de type **App\Repositories\PhotosRepository**. On utilise la méthode **save** de la classe ainsi injectée pour faire le traitement.

De cette façon le contrôleur ignore totalement comment se fait la gestion, il sait juste que la classe **PhotosRepository** sait la faire. Il se contente d'utiliser la méthode de cette classe qui est « injectée ».

Maintenant vous vous demandez sans doute comment cette classe est injectée là, en d'autres termes comment et où est créée cette instance. Et bien Laravel est assez malin pour le faire lui-même.

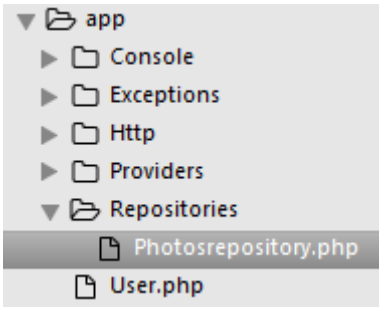
PHP est très tolérant sur les types des variables (mais ça s'est quand même bien amélioré avec la version 7 !). Lorsque vous en déclarez une vous n'êtes pas obligé de préciser que c'est une string ou un array. PHP devine le type selon la valeur affectée. Il en est de même pour les paramètres des fonctions. Mais personne ne vous empêche de déclarer un type comme je l'ai fait ici pour le paramètre de la méthode. C'est même indispensable pour que Laravel

sache quelle classe est concernée. Étant donné que je déclare le type, Laravel est capable de créer une instance de ce type et de l'injecter dans le contrôleur.

Pour trouver la classe, Laravel utilise l'introspection (**reflexion** en anglais) de PHP qui permet d'inspecter le code en cours d'exécution. Elle permet aussi de manipuler du code et donc de créer par exemple un objet d'une certaine classe. Vous pouvez trouver tous les renseignements **dans le manuel PHP**.

XI-B - La gestion

Maintenant qu'on a dit au contrôleur qu'une classe s'occupe de la gestion il nous faut la créer. Pour bien organiser notre application on crée un nouveau dossier et on place notre classe dedans :



Le codage ne pose aucun problème parce qu'il est identique à ce qu'on avait dans le contrôleur :

```
1. <?php
2.
3. namespace App\Repositories;
4.
5. use Illuminate\Http\UploadedFile;
6.
7. class PhotosRepository
8. {
9.     public function save(UploadedFile $image)
10.     {
11.         $image->store(config('images.path'), 'public');
12.     }
13. }
```

Attention à ne pas oublier les espaces de noms !

Maintenant notre code est parfaitement organisé et facile à maintenir et à tester.

Mais allons un peu plus loin, créons une interface pour notre classe :

```
1. <?php
2.
3. namespace App\Repositories;
4.
5. use Illuminate\Http\UploadedFile;
6.
7. interface PhotosRepositoryInterface
8. {
9.   public function save(UploadedFile $image);
10. }
```

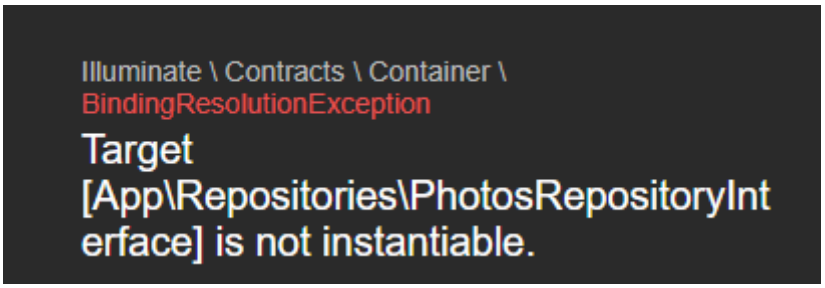
Il suffit ensuite d'en informer la classe **PhotosRepository** :

```
class PhotosRepository implements PhotosRepositoryInterface
```

Ce qui serait bien maintenant serait dans notre contrôleur de référencer l'interface :

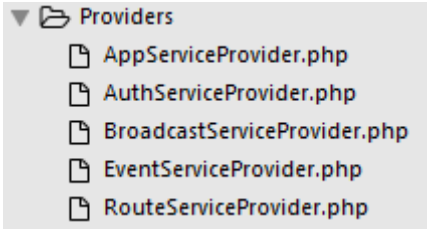
```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests\ImagesRequest;
use App\Repositories\PhotosRepositoryInterface;

class PhotoController extends Controller
{
    public function create()
    {
        return view('photo');
    }

    public function store(ImagesRequest $request,
 PhotosRepositoryInterface $photosRepository)
    {
        $photosRepository->save($request->image);

        return view('photo_ok');
    }
}
```

Le souci c'est que Laravel n'arrive pas à deviner la classe à instancier à partir de cette interface :



Comment s'en sortir ?

Lorsque j'ai présenté la structure de Laravel j'ai mentionné la présence de *providers* :



À quoi sert un provider ? Tout simplement à procéder à des initialisations : événements, middlewares, et surtout des liaisons de dépendance. Laravel possède un conteneur de dépendances qui constitue le cœur de son fonctionnement. C'est grâce à ce conteneur qu'on va pouvoir établir une liaison entre une interface et une classe.

Ouvrez le fichier **app\Providers\AppServiceProvider.php** et ajoutez cette ligne de code :

```php
public function register()
{
    $this->app->bind(
        'App\Repositories\PhotosRepositoryInterface',
        'App\Repositories\PhotosRepository'
    );
}
```

La méthode **register** est activée au démarrage de l'application, c'est l'endroit idéal pour notre liaison. Ici on dit à l'application (**app**) d'établir une liaison (**bind**) entre l'interface **App\Repositories\PhotosRepositoryInterface** et la classe **App\Repositories\PhotosRepository**. Ainsi chaque fois qu'on se référera à cette interface dans une injection Laravel saura quelle classe instancier. Si on veut changer la classe de gestion il suffit de modifier le code du provider.

Maintenant notre application fonctionne.

Si vous obtenez encore un message d'erreur vous disant que l'interface ne peut pas être instanciée lancez la commande :

```
composer dumpautoload
```

XI-C - Les façades

Laravel propose de nombreuses façades pour simplifier la syntaxe. Vous pouvez les trouver déclarées dans le fichier **config/app.php** :

```
1. 'aliases' => [
2.
3.     'App' => Illuminate\Support\Facades\App::class,
4.     'Artisan' => Illuminate\Support\Facades\Artisan::class,
5.     'Auth' => Illuminate\Support\Facades\Auth::class,
6.     'Blade' => Illuminate\Support\Facades\Blade::class,
7.     'Broadcast' => Illuminate\Support\Facades\Broadcast::class,
8.     'Bus' => Illuminate\Support\Facades\Bus::class,
9.     'Cache' => Illuminate\Support\Facades\Cache::class,
10.     'Config' => Illuminate\Support\Facades\Config::class,
11.     'Cookie' => Illuminate\Support\Facades\Cookie::class,
12.     'Crypt' => Illuminate\Support\Facades\Crypt::class,
13.     'DB' => Illuminate\Support\Facades\DB::class,
14.     'Eloquent' => Illuminate\Database\Eloquent\Model::class,
15.     'Event' => Illuminate\Support\Facades\Event::class,
16.     'File' => Illuminate\Support\Facades\File::class,
17.     'Gate' => Illuminate\Support\Facades\Gate::class,
18.     'Hash' => Illuminate\Support\Facades\Hash::class,
19.     'Lang' => Illuminate\Support\Facades\Lang::class,
20.     'Log' => Illuminate\Support\Facades\Log::class,
21.     'Mail' => Illuminate\Support\Facades\Mail::class,
22.     'Notification' => Illuminate\Support\Facades\Notification::class,
23.     'Password' => Illuminate\Support\Facades\Password::class,
24.     'Queue' => Illuminate\Support\Facades\Queue::class,
25.     'Redirect' => Illuminate\Support\Facades\Redirect::class,
26.     'Redis' => Illuminate\Support\Facades\Redis::class,
27.     'Request' => Illuminate\Support\Facades\Request::class,
28.     'Response' => Illuminate\Support\Facades\Response::class,
29.     'Route' => Illuminate\Support\Facades\Route::class,
30.     'Schema' => Illuminate\Support\Facades\Schema::class,
31.     'Session' => Illuminate\Support\Facades\Session::class,
32.     'Storage' => Illuminate\Support\Facades\Storage::class,
33.     'URL' => Illuminate\Support\Facades\URL::class,
34.     'Validator' => Illuminate\Support\Facades\Validator::class,
35.     'View' => Illuminate\Support\Facades\View::class,
36.
37. ],
```

Vous trouvez dans ce tableau le nom de la façade et la classe qui met en place cette façade. Par exemple pour les routes on a la façade **Route** qui correspond à la classe **Illuminate\Support\Facades\Route**. Regardons cette classe :

```
1. <?php
2.
3. namespace Illuminate\Support\Facades;
4.
5. /**
6.  * @see \Illuminate\Routing\Router
```

```
7.  */
8. class Route extends Facade
9. {
10.     /**
11.      * Get the registered name of the component.
12.      *
13.      * @return string
14.      */
15.     protected static function getFacadeAccessor()
16.     {
17.         return 'router';
18.     }
19. }
```

On se contente de retourner **router**. Il faut aller voir dans le fichier **Illuminate\Routing\RoutingServiceProvider** pour trouver l'enregistrement du router :

```
1. protected function registerRouter()
2. {
3.     $this->app->singleton('router', function ($app) {
4.         return new Router($app['events'], $app);
5.     });
6. }
```

Les providers permettent d'enregistrer des composants dans le conteneur de Laravel. Ici on déclare **router** et on voit qu'on crée une instance de la classe **Router** (**new Router**…). Le nom complet est **Illuminate\Routing\Router**. Si vous allez voir cette classe vous trouverez les méthodes qu'on a utilisées dans ce chapitre, par exemple **get** :

```
1. public function get($uri, $action = null)
2. {
3.     return $this->addRoute(['GET', 'HEAD'], $uri, $action);
4. }
```

Autrement dit si j'écris en utilisant la façade :

```
Route::get('/', function() { return 'Coucou'; });
```

J'obtiens le même résultat que si j'écris en allant chercher le routeur dans le conteneur :

```
$this->app['router']->get('/', function() { return 'Coucou'; });
```

Ou encore en utilisant un helper :

```
app('router')->get('/', function() { return 'Coucou'; });
```

La différence est que la première syntaxe est plus simple et intuitive mais certains n'aiment pas trop ce genre d'appel statique.

XI-D - En résumé

- Un contrôleur doit déléguer toute tâche qui ne relève pas de sa compétence.
- L'injection de dépendance permet de bien séparer les tâches, de simplifier la maintenance du code et les tests unitaires.
- Les providers permettent de faire des initialisations, en particulier des liaisons de dépendance entre interfaces et classes.
- Laravel est équipé de nombreuses façades qui simplifient la syntaxe.
- Il existe aussi des helpers pour simplifier la syntaxe.